Chapter 1 - Getting Started

1.1 Introduction

Ordinary differential equations are ubiquitous in science and engineering, especially when mathematically modelling how physical systems evolve.

Many courses on ODEs focus on analytical solution techniques.

Unfortunately these techniques often cannot handle the large, complicated, and nonlinear systems of ODEs that arise in practice.

This course is about solving ODEs numerically.

Along the way, we will also touch on topics such as automatic differentiation, quadrature, and the solution of partial differential equations.

We emphasize that **any use of numerics should be complementary to analysis**: numerical algorithms and software should be based on solid theory, and it is generally ill-advised to attempt to numerically solve a problem for which you have done no analysis.

We study numerical methods to understand what is meant by a numerical solution and what can reasonably be expected from a numerical method and/or software.

In this course we emphasize the use of *realistic problems* as examples; toy problems only take you so far in life.

We make good use of the popular *problem-solving environment* (PSE) MATLAB. However, alternate environments, in particular MAPLE, will also be used.

Review of ODEs

An ODE represents a relationship between a function and its rates of change; e.g.,

$$\dot{y}(t) = y(t), \qquad 0 \leq t \leq 10.$$

This does not completely specify the solution.

Often (but not always!) solutions are specified by means of an initial value; e.g., $y(0) = 1$.

In this case, the ODE has the unique solution $y(t) = e^t$.

This is called an *initial-value problem* (IVP).

Many IVPs in practice have unique solutions.

Conditions on the solution can be specified in a more complicated way.

Consider

$$y''(x) + y(x) = 0, \qquad 0 \le x \le b.$$

A solution to this ODE may be specified by conditions at both ends of the interval; e.g., $y(0) = 0$, $y(b) = 0$.

This is called a *boundary-value problem* (BVP).

This BVP always has the trivial solution $y(x) \equiv 0$.

But for certain values of b, this BVP has infinitely many solutions; e.g., $b = 2\pi$, $y(x) = c \sin x$ is a solution for any constant c.

In general, a BVP may have no solution; e.g., if $b = 2\pi$ and $y(b) \neq 0$.

With a BVP, anything is possible!

A comment on analytical solutions

It is easy to think that one should always use an analytical solution (whenever one exists) over a numerical solution.

This may not be true 100% of the time.

For example, small changes to a problem can dramatically change its analytical solution.

See `dsolveDemo.m`

Even simple-looking problems may not have a solution expressible in terms of any familiar functions.

(If Maple cannot find an analytical solution to your ODE, it usually means that one does not exist.)

Systems of ODEs rarely have explicit solutions!

Analytical vs. numerical

Analytical solutions provide a lot of insight, but often we have to turn to plotting software to plot particular solutions anyway.

So why bother with an analytical solution at all?

It depends on what you want from a solution.

Analytical solutions can give you insight on things such as asymptotic behaviour of solutions as $t \to \infty$, singular behaviour, or dependence on initial conditions or other parameters.

These things are hard to elicit from numerical solutions because each numerical solution is an isolated solution to a problem with a particular choice of parameters, time interval, etc.

On the other hand, numerical solutions to all the problems in `dsolveDemo.m` are obtained with ease.

We often combine analytical and numerical techniques to solve a given problem.

1.2 Existence, Uniqueness, and Well-Posedness

This is not about being fussy.

These things tell you whether in fact you can solve a given problem, and if so, how well.

We will see problems that have no solution. In this case, we expect trouble!

We will also see problems with more than one solution. In this case, we can expect some trouble computing the "correct" one.

We note that although theory can guarantee the existence of a unique solution, there is no substitute for an understanding of the phenomena being modeled.

Existence and uniqueness are much simpler for IVPs.

We study *systems* of *explicit* ODEs of size m:

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)). \tag{1a}$$

$\mathbf{y}(t)$ is really a vector of size m; i.e.,

$$\mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{bmatrix}.$$

The same goes for the vector $\mathbf{f}(t, \mathbf{y}(t))$.

An IVP is defined by having all the values of $\mathbf{y}$ known at some "initial" point; without loss of generality, we take this point to be $t = 0$:

$$\mathbf{y}(0) = \mathbf{y}_0. \tag{1b}$$

Roughly speaking, if $\mathbf{f}(t, \mathbf{y})$ is smooth for all $(t, \mathbf{y})$ in a region $\mathbf{R}$ containing the initial data $(0, \mathbf{y}_0)$, then the IVP (1) has a unique solution. This takes care of most IVPs in practice.

Note 1. *We are guaranteed by theory that the solution extends to the boundary of* $\mathbf{R}$.

However, this does not mean the solution for given initial data $(0, \mathbf{y}_0)$ *exists throughout an interval* $t \in [a, b]$ *that is completely contained in* $\mathbf{R}$*!*

What does this mean?

Consider the IVP

$$\dot{y} = y^2, \qquad y(0) = 1.$$

The right-hand side $f(t, y) = y^2$ is smooth everywhere in (t, y); i.e.,

$$\mathbf{R} = \{-\infty < t < \infty,\ -\infty < y < \infty\}.$$

You might think the solution exists for all t for this problem (or any IVP for this ODE).

However, the unique solution for this problem is

$$y(t) = \frac{1}{1-t}.$$

So the solution that starts at $(0, 1)$ exits the top of $\mathbf{R}$ as $t \to 1^-$; *it does not exist for all t!*

This kind of thing does happen for real problems too!

It is usually reasonable to ask that a numerical approximation do a good job until the solution becomes too large for the computer arithmetic used.

Both existence and uniqueness can be problematic for *implicit* ODEs

$$\mathbf{F}(t, \mathbf{y}(t), \dot{\mathbf{y}}(t)) = \mathbf{0}.$$

For example, obviously the ODE

$$[\dot{\mathbf{y}}(t)]^2 + 1 = 0$$

has no real solutions.

As another example of an implicit ODE, consider now how the solutions $\mathbf{y}$ of a system of *algebraic equations*

$$\mathbf{F}(\mathbf{y}, \lambda) = \mathbf{0}$$

depends on a (scalar) parameter λ.

Taking $d/d\lambda$ by using the Chain Rule, we get

$$\frac{\partial \mathbf{F}}{\partial \mathbf{y}} \frac{d\mathbf{y}}{d\lambda} + \frac{\partial \mathbf{F}}{\partial \lambda} = \mathbf{0}.$$

This is an implicit system of first-order ODEs. If we can find one solution $\mathbf{y}_0$ for some parameter value λ_0 (i.e., $\mathbf{F}(\mathbf{y}_0, \lambda_0) = \mathbf{0}$) then we can formulate an IVP for $\mathbf{y} = \mathbf{y}(\lambda)$.

If the Jacobian matrix

$$\mathbf{J} := \frac{\partial \mathbf{F}}{\partial \mathbf{y}}$$

is non-singular, we can write the ODEs explicitly as

$$\frac{d\mathbf{y}}{d\lambda} = -\mathbf{J}^{-1}\frac{\partial \mathbf{F}}{\partial \lambda}.$$

Problems occur when $\mathbf{J}$ is singular, and this happens for many interesting problems!

In such cases we say that solutions *bifurcate*; i.e., the number of solutions changes.

At this point we would need **analysis plus software** to sort out the behaviour of the solutions near this point.

Suppose we want to compute steady-state solutions of the ODE

$$\dot{y} = y^2 - \lambda.$$

This corresponds to solving

$$F(y, \lambda) := y^2 - \lambda = 0.$$

If $\lambda \geq 0$, one steady-state solution is $y(\lambda) = \sqrt{\lambda}$.

To study the general dependence of the steady state on λ, we could compute it as the solution of the IVP

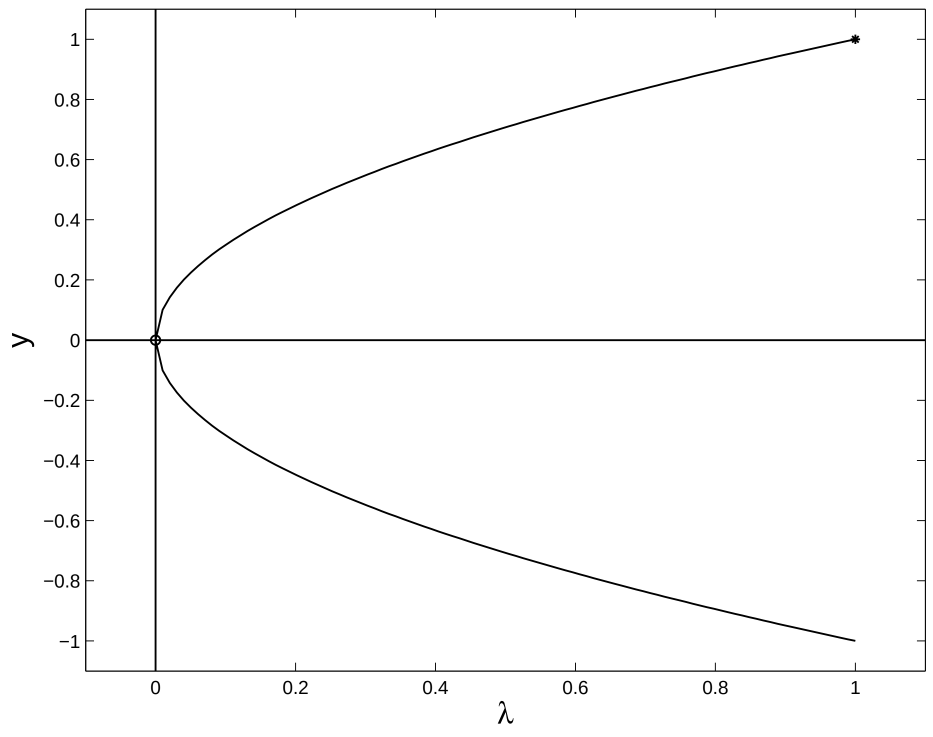
$$2y\frac{dy}{d\lambda} - 1 = 0, \qquad y(1) = 1. \tag{2}$$

If $y \neq 0$, we can write the ODE in explicit form and solve for values of λ decreasing from 1.

But this ODE is singular when $y = 0$, corresponding in this case to $\lambda = 0$.

So the singular point $(0, 0)$ means there may be more than one solution to the IVP passing through this point; in fact we know $y(\lambda) = -\sqrt{\lambda}$ is another solution.

We can solve the IVP (2) using standard software, but we will run into trouble as we approach $(0,0)$.



Technical point: Lipschitz condition

For future reference, we define more precisely what we mean by “smooth” $\mathbf{f}(t, \mathbf{y})$.

1. $\mathbf{f}(t, \mathbf{y})$ is continuous in $\mathbf{R}$
2. Sufficiently many $\mathbf{y}$ derivatives of $\mathbf{f}$ are also continuous.

Technically, $\mathbf{f}$ must satisfy a *Lipschitz condition*; i.e., there exists a constant L such that for all pairs $(t, \mathbf{y}_1)$ and $(t, \mathbf{y}_2)$ in $\mathbf{R}$, we have

$$\|\mathbf{f}(t, \mathbf{y}_1) - \mathbf{f}(t, \mathbf{y}_2)\| \leq L\|\mathbf{y}_1 - \mathbf{y}_2\|.$$

For single equations, the Mean Value Theorem states that

$$f(t, y_1) - f(t, y_2) = \frac{\partial f}{\partial y}(t, \hat{y})(y_1 - y_2),$$

where $\hat{y}$ lies in the open interval with endpoints y_1 and y_2. So f satisfies a Lipschitz condition if

$$\left|\frac{\partial f}{\partial y}\right| \leq L \qquad \forall\ (t, y) \in \mathbf{R}.$$

For vectors $\mathbf{f}$, we must find a bound such as this for all first partial derivatives of $\mathbf{f}$.

In some sense we are applying the scalar Mean Value Theorem to each component of $\mathbf{f}$ for each component of $\mathbf{y}$.

Each application will generally involve a different mean value $\hat{y}$, but this fact is often only implied in print.

Well-posedness

Roughly speaking, a problem is *well-posed* if small changes to the data lead to small changes in the solution.

Such a problem is also said to be *well-conditioned* or *insensitive* to perturbation.

This has to be a property of the problem if we want to solve it (directly) on a computer.

The numerical methods that we study can be viewed as the exact solution to a problem that is "close" to the one we wish to solve.

This is the best one can hope for with finite arithmetic.

Consider the motion of a *simple pendulum*:

$$\ddot{\theta}(t) + \sin\theta = 0, \qquad \theta(0) = 0,\ \dot{\theta}(0) = \omega_0,$$

where $\theta(t)$ is the angle the pendulum makes with the vertical at time t.

If $\omega_0 = 0$, the pendulum does not move.

If ω_0 is small enough, the pendulum will *oscillate* forever.

If ω_0 is large enough, the pendulum will *revolve* forever.

Somewhere in between there is a magical value ω_0^* such that the pendulum will approach a vertical configuration.

Clearly this solution is *unstable*: any perturbation of ω_0^* leads to a very different type of solution.

The IVP with $\dot{\theta}(0) = \omega_0^*$ is ill-posed (ill-conditioned, sensitive to perturbation) on sufficiently long time intervals.

On physical grounds, one can deduce that the ill-posed solution satisfies

$$\theta(\infty) = \pi \qquad \text{and} \qquad \dot{\theta}(\infty) = 0.$$

Using conservation of energy, we find that $\omega_0^* = 2$.

Thus, the IVP

$$\ddot{\theta}(t) + \sin\theta = 0, \quad \theta(0) = 0,\ \dot{\theta}(0) = 2,$$

is unstable.

What happens if we try to solve it anyway using, e.g., MATLAB's `ode45`?

See `unstableIVP.m`

This unstable solution is more naturally described by a BVP:

$$\ddot{\theta}(t) + \sin\theta = 0, \quad \theta(0) = 0,\ \theta(\infty) = \pi.$$

Note 2. *When formulating a problem as a BVP, it is not always clear what BCs to use!*

e.g., why not use $\dot{\theta}(\infty) = 0$?

One problem with this BC is that besides the solution we want, it also admits (at least) one other solution; i.e., $\theta(t) \equiv 0$.

Finding a non-trivial solution when a trivial one exists can be challenging!

But this BVP is defined on an infinite interval[1]; existence, uniqueness, and well-posedness are not as clear anymore! Besides, all the software we discuss is meant for problems on finite intervals.

One way to solve such a problem is to *truncate* the infinite interval at a (sufficiently large) finite point; this point is idealized as being at infinity.

See `stableBVP.m`

Note 3. *1. This approach does not always work!*

2. One should solve a sequence *of problems, with increasing interval size, to validate the computed solution. The* MATLAB *function* `bvpxtend` *helps.*

For this example, we could have gotten away with a much smaller interval (as small as $[0, 7]$).

Also, the solution would have improved upon lowering the tolerances on the solution to the IVP.

[1]It is not unusual for physical problems to be defined this way.

Another example: Projectile problem

Consider the *planar*[2] motion of a projectile fired from a cannon. The equations of motion are

$$
\begin{aligned}
y' &= \tan\phi, \\
v' &= -\frac{g\sin\phi + \nu v^2}{v\cos\phi}, \\
\phi' &= -\frac{g}{v^2},
\end{aligned}
$$

where y is the height of the projectile above the cannon, v is the velocity of the projectile, and ϕ is the angle (in radians) of the projectile with respect to the horizontal.

The independent variable x measures the horizontal distance of the projectile from the cannon.

The constant ν represents air resistance, and $g = 0.032$ is the (appropriately scaled) gravitational constant.

[2]We ignore 3D effects such as cross winds or rotation.

We take $y(0) = 0$ and assume that $v(0) = v_0$ is given.

The usual projectile problem is to choose the cannon elevation $\phi(0)$ so that $y(b) = 0$.

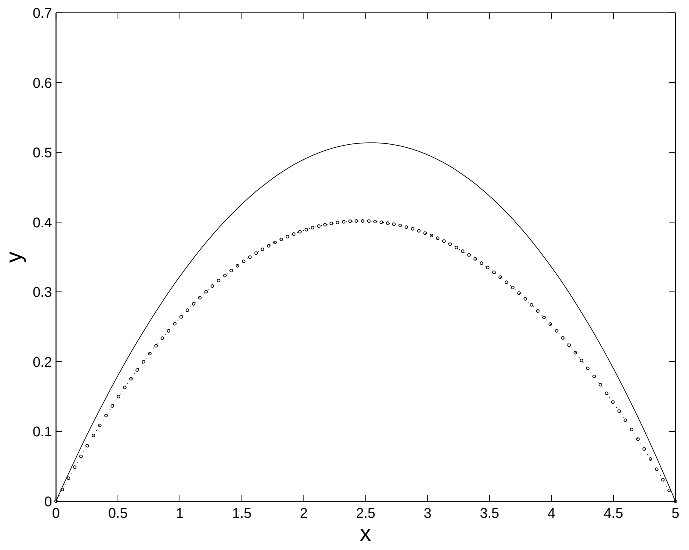
Note 4. *There are 3 BCs corresponding to 3 first-order derivatives in the system of ODEs.*

Does this problem have a solution?

Not when b is outside of the cannon's range!

When b is exactly equal to the cannon's range, there is exactly one solution.

If b is within the cannon's range, there are two solutions (one which is more direct than the other).



Existence and uniqueness questions are much easier to answer for IVPs than for BVPs.

Theoretical results about uniqueness and number of solutions for BVPs exist, but they are often so narrow that they are rarely used in practice.

That is why it is of great help to have some understanding of what things should look like or how they should behave.

Determining the exact number of solutions is very difficult; usually the best we can do is find a solution that is “close” to some initial guess.

There is a real possibility of obtaining a “wrong” (i.e., unintended) solution when solving BVPs!

Stability

Stability is key to understanding how numerical methods behave when solving (1).

Numerical methods typically produce approximations $\mathbf{y}_n \approx \mathbf{y}(t_n)$ on a mesh

$$0 = t_0 < t_1 < t_2 < \ldots < t_N = b$$

that is chosen by the solver; i.e., the integration starts with $\mathbf{y}_0$ and in general takes steps of size $\Delta t_n = t_n - t_{n-1}$ to obtain $\mathbf{y}_n$.

What the solver actually does is not what you might expect!

First we define the *local solution* $\tilde{\mathbf{y}}(t)$ as the solution to the (local) IVP

$$\dot{\tilde{\mathbf{y}}} = \mathbf{f}(t, \tilde{\mathbf{y}}), \qquad \tilde{\mathbf{y}}(t_n) = \mathbf{y}_n;$$

i.e., this problem takes $\mathbf{y}_n$ as the *exact* initial value.

We call it a local IVP because we imagine taking only one step with it.

The solver actually tries to find $\mathbf{y}_{n+1}$ such that the *local error*

$$\tilde{\mathbf{y}}(t_{n+1}) - \mathbf{y}_{n+1}$$

does not exceed error tolerances set by the user.

However the *global* (or "true") error is

$$\mathbf{y}(t_{n+1}) - \mathbf{y}_{n+1};$$

i.e., **solvers do not control global error (directly)!**

How the error propagates can be analyzed by writing

$$\mathbf{y}(t_{n+1}) - \mathbf{y}_{n+1} = \mathbf{y}(t_{n+1}) - \tilde{\mathbf{y}}(t_{n+1}) + \tilde{\mathbf{y}}(t_{n+1}) - \mathbf{y}_{n+1}$$

The second difference is the local error.

The first difference is the difference at t_{n+1} of two solutions of the ODE (1a) *with different "initial" values at* t_n: one starts at $\mathbf{y}(t_n)$; the other starts at $\mathbf{y}_n$.

This term is a characteristic of the ODE, so it cannot be controlled by the solver.

If the IVP is unstable (i.e., neighbouring solutions rapidly spread apart), then we see that **global errors can be large even if local errors are small**.

On the other hand, if the IVP is stable (i.e., neighbouring solutions come together), then global errors will be comparable to local errors.

This gives us a fundamental limitation on all numerical methods that we consider: **if the IVP is unstable, no matter how well you control the local error, the global error will eventually become large**.

How quickly this actually happens depends on how unstable the problem is and how accurately the local error is controlled.

Example

Consider the IVP

$$\dot{y} = 5(y - t^2), \qquad y(0) = 0.08.$$

Ignoring the initial condition, the exact solution is

$$y(t) = t^2 + 0.4t + 0.08 + Ce^{5t}.$$

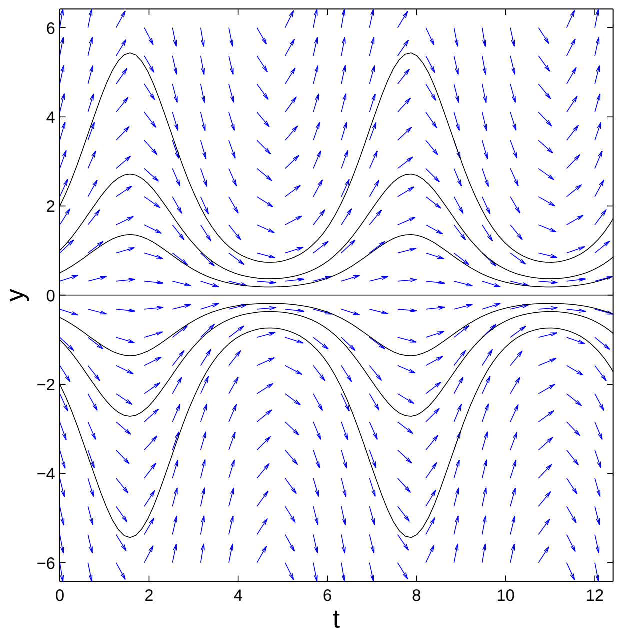
This IVP is unstable because neighbouring solutions (with constants C_1 and C_2) differ by $(C_1 - C_2)e^{5t}$; i.e., they exponentially diverge from each other!

So even if we had a quasi-magical numerical method that *only made a tiny error in its first step* (and was exact the rest of the time), the error in the quasi-magical numerical solution would grow exponentially and eventually result in an unacceptable numerical solution $\mathbf{y}_n$.

Stable vs. unstable

Consider the direction field and solution curves for the ODE

$$\dot{y} = (\cos t)y. \tag{3}$$



We can see solution curves diverging in some regions and converging in others.

This example shows that it is an over-simplification to simply say a problem is stable or unstable!

Note 5. *Stability depends on direction; i.e., some problems are unstable when solved forward in time but stable when solved backward in time.*

Note 6. *It is very possible that for systems of ODEs you can have one component be stable while another be unstable at the same time. In such cases the coupling of the components to each other (and then to the solver!) may make the overall behaviour very complicated to sort out.*

A numerical experiment

The simplest method for the numerical solution of IVPs is *Euler's method*[3].

Starting from a given approximation $\mathbf{y}_n$ at time $t = t_n$, it uses a fixed step size Δt and marches the approximate solution forward according to the formula

$$\begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + \Delta t\, \mathbf{f}(t_n, \mathbf{y}_n), \\ t_{n+1} &= t_n + \Delta t, \qquad n = 0, 1, 2, \ldots. \end{aligned}$$

This method has a nice geometric interpretation.

We can imagine that $\mathbf{f}(t, \mathbf{y})$ specifies a *velocity field* for $\mathbf{y}$.

Then Euler's method can be thought of as sampling the velocity field at the point $(t_n, \mathbf{y}_n)$, assuming it is constant, and following it for a length of time Δt.

[3]Those in the biz also call it *forward Euler*.

This amounts to following the *tangent line* to the velocity field at $(t_n, \mathbf{y}_n)$ for some length of time Δt.

This procedure is repeated starting from $(t_{n+1}, \mathbf{y}_{n+1})$ to approximate $\mathbf{y}_{n+2}$, etc.

See `forwardEulerDemo.m` and `forwardEulerDemo2.m`

Given initial time $t = 0$, initial value $\mathbf{y}_0$, final time $t = b$, right-hand side function $\mathbf{f}(t, \mathbf{y})$, and step size `dt`, some MATLAB code to implement Euler's method might look like:

```
t = 0;
y = y0;
while t <= b
  y = y + dt*feval(f,t,y)
  t = t + dt
end
```

As you can imagine, the answer gets more accurate as you reduce Δt; but of course more work has to be done to get to $t = b$.

Note: There is no inherent need to assume Δt is a constant.

In fact if we could estimate how big a step size Δt_{n+1} starting from t_n we could get away with and still satisfy the given tolerance, then we could take that instead.

This generally yields a more efficient algorithm than one with fixed step size.

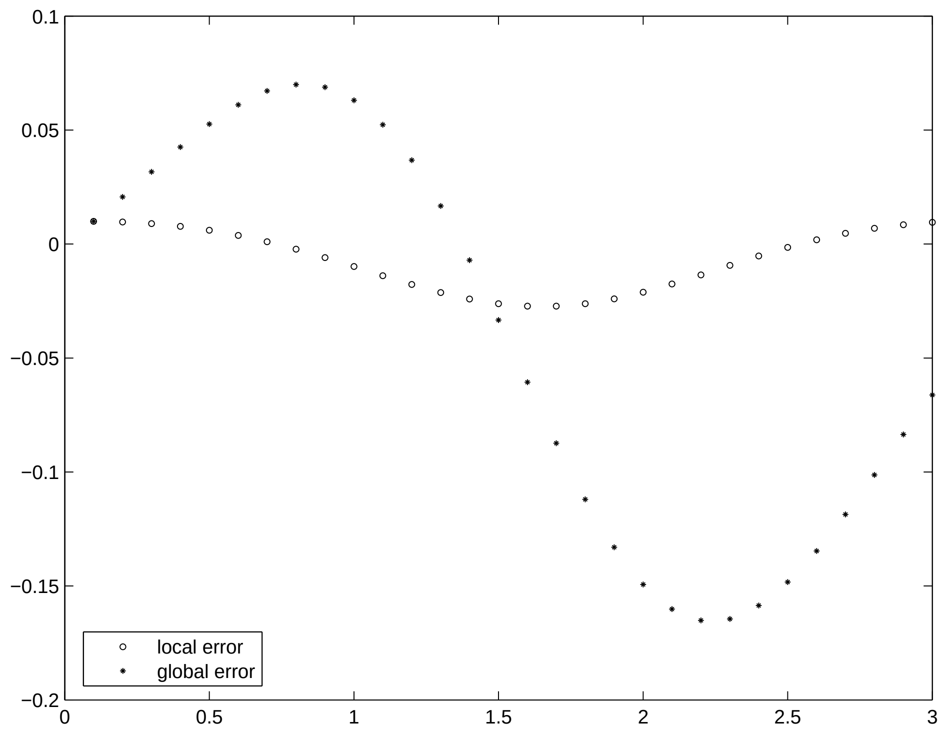
Consider now the solution to (3) with IC $y(0) = 2$:

$$y(t) = 2e^{\sin t}.$$

The local solution starting from the point (t_n, y_n) is

$$\tilde{y}(t) = y_n e^{(\sin t - \sin t_n)}.$$

We now plot the local and global errors when Euler's method with a constant step size $\Delta t = 0.1$ is used to integrate (3) from $t = 0$ to $t = 3$.



For this problem, we notice that the local error is never very large.

This is somewhat fortunate because we are not making any attempt to control it!

By definition, the local and global errors are the same after the first step.

After that, the global errors grow or decay according to the stability of the problem.

Backward error analysis

This is a powerful tool for studying the errors produced by a numerical method.

The (usual) method of *forward error analysis* is that you somehow keep track of all the errors you make as you step through your computational procedure. At the end of the day, you bound all these errors.

Conversely, *backward error analysis* looks at the numerical solution as the *exact solution* to a *perturbed problem*. Then one asks how far is the perturbed problem from the original one.

All the MATLAB solvers produce piecewise-smooth numerical solutions $S(t)$ on the *whole interval* $[0, b]$.

As a simple case, you can imagine connecting the output values $\mathbf{y}_n$ and $\mathbf{y}_{n+1}$ with a straight line and taking that line[4] to be the solution *everywhere* in $[t_n, t_{n+1}]$.

[4]This line is also called a *linear interpolant*.

Usually the $\mathbf{y}_i$ are known to higher accuracy than second order, and so we will have to do something more sophisticated than simple linear interpolation for the approximations within the interval to be as accurate as those at the end points.

The *residual* $\mathbf{r}(t)$ of such an approximation is the amount by which the $\mathbf{S}(t)$ fails to satisfy the ODE:

$$\mathbf{r}(t) := \dot{\mathbf{S}}(t) - \mathbf{f}(t, \mathbf{S}(t)).$$

In other words, $\mathbf{S}(t)$ is the *exact solution* to the *perturbed* ODE

$$\dot{\mathbf{S}}(t) = \mathbf{f}(t, \mathbf{S}(t)) + \mathbf{r}(t).$$

So in the sense of backward error analysis, $\mathbf{S}(t)$ is a good approximation if it satisfies an ODE that is “close” to the original; i.e., $\|\mathbf{r}(t)\|$ is small.

Not only is this a perfectly fine definition of what constitutes a "good" solution, but also if the IVP is well-posed, then $\|\mathbf{r}(t)\|$ small plus well-posedness implies $\mathbf{S}(t)$ is "close" to $\mathbf{y}(t)$ — the usual definition of a "good" solution.

In other words, in general **a solver tries to produce an approximate solution with a small residual**.

MATLAB's BVP solver `bvp4c` does this directly; IVP solvers such as `ode45` or `ode15s` do it indirectly.

In the case of MATLAB's newest BVP solver `bvp5c`, there is a remarkable relationship between the residual and the global error, so in this case control of the global error by means of residual control is more direct than usual.

1.3 Standard Form

ODEs are not always kind enough to come in a form ready for software to solve.

ODEs are most commonly accepted by IVP solvers in the form of *systems of first-order equations*:

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)).$$

The MATLAB IVP solvers accept the more general form

$$\mathbf{M}(t, \mathbf{y})\dot{\mathbf{y}}(t) = \mathbf{F}(t, \mathbf{y}(t)),$$

where $\mathbf{M}(t, \mathbf{y})$ is a **non-singular**[5] *mass matrix*. Of course we could write $\mathbf{f}(t, \mathbf{y}) := \mathbf{M}(t, \mathbf{y})^{-1}\mathbf{F}(t, \mathbf{y})$, but sometimes it is more convenient and/or more efficient to not do this.

[5] If $\mathbf{M}$ is singular, then (at least locally) we have a *differential-algebraic equation* (DAE).

The usual way to convert higher-order ODEs to first order is by introducing new dependent variables to represent all the old dependent variables and their derivatives.

It is easy to see what to do by example.

Consider the *two-body problem*.

This describes the orbit of one body under the gravitational attraction of a much heavier body (like a planet orbiting a sun).

With the origin at the centre of the “sun”, the Cartesian coordinates $u(t)$ and $v(t)$ of the “planet” satisfy

$$\begin{aligned} \ddot{u}(t) &= -\frac{u(t)}{[u^2(t)+v^2(t)]^{3/2}}, \\ \ddot{v}(t) &= -\frac{v(t)}{[u^2(t)+v^2(t)]^{3/2}}. \end{aligned}$$

To convert to a first-order system of ODEs, we define

$$\mathbf{y}(t) = (u(t),\ \dot{u}(t),\ v(t),\ \dot{v}(t))^T.$$

Then

$$\dot{\mathbf{y}}(t) = \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \\ \dot{y}_3(t) \\ \dot{y}_4(t) \end{bmatrix} = \begin{bmatrix} y_2(t) \\ -\frac{y_1(t)}{[y_1^2(t)+y_3^2(t)]^{3/2}} \\ y_4(t) \\ -\frac{y_3(t)}{[y_1^2(t)+y_3^2(t)]^{3/2}} \end{bmatrix}.$$

A MATLAB script to define this might look like

```
function ydot = twobody(t,y)
% define this variable for efficiency
r3 = (y(1)^2 + y(3)^2)^(3/2);
ydot = [y(2); -y(1)/r3; y(4); -y(3)/r3];
```

The two-body problem and the pendulum problem are derived from Newton's laws of motion, hence the second derivatives.

If there is no dissipation, then there are no terms involving first derivatives. This leads to the form

$$\ddot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}),$$

with initial position $\mathbf{y}(0)$ and initial velocity $\dot{\mathbf{y}}(0)$ given.

Of course we could write this as a first-order system, but there are efficient numerical methods[6] that make it worth dealing with the second-order form directly.

Note 7. *There are similar methods that deal directly with the form*

$$\ddot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}, \dot{\mathbf{y}})$$

(or even higher-order generalizations of this) but there seem to be no computational advantages to do this in the IVP context. There are computational advantages for BVPs, however.

[6]called *Runge–Kutta–Nyström methods*

A very useful trick is to introduce additional unknowns (including constants) in order to compute quantities related to the solution.

Consider the Sturm–Liouville eigenvalue problem

$$y''(x) + \lambda y(x) = 0, \quad y(0) = 0,\ y(2\pi) = 0. \tag{4}$$

For special values of λ (known as *eigenvalues*) there are non-trivial solutions $y(x)$ (known as *eigenfunctions*).

As it stands, (4) only defines $y(x)$ up to a multiplicative constant. We can determine a unique solution by normalizing the solution by the condition

$$\int_0^{2\pi} y^2(x)\,dx = 1. \tag{5}$$

A convenient way to impose (5) is by introducing the variable

$$y_3(x) = \int_0^x y^2(x')\,dx'.$$

We also need to note that λ is unknown, so we introduce the variable

$$y_4(x) = \lambda.$$

Defining as usual $y_1(x) = y(x)$ and $y_2(x) = y'(x)$, we can write (4) in standard form as

$$\begin{aligned} y_1' &= y_2, \\ y_2' &= -y_4 y_1, \\ y_3' &= y_1^2, \\ y_4' &= 0, \end{aligned}$$

subject to

$$y_1(0) = 0,\ y_1(2\pi) = 0,\ y_3(0) = 0,\ y_3(2\pi) = 1.$$

Note 8. MATLAB*'s BVP solvers* `bvp4c` *and* `bvp5c` *accept problems with unknown parameters, but most other BVP solvers require the problem to be in standard form as just described.*

1.4 Control of the Error

Error control is important for both efficiency and reliability.

The more accuracy you want, the more the computation will cost.

Solvers attempt to meet the accuracy requested with the minimum amount of computational effort.

Solvers can request two kinds of tolerances: a scalar relative tolerance `RelTol` and a vector of absolute tolerances `AbsTol`.

To simplify matters, some codes only accept a value for `RelTol`.

Also, if a scalar is supplied for `AbsTol`, it is applied as a vector with the same value for each component.

It is also required that $\texttt{RelTol}, \texttt{AbsTol}_i > 0$ for all i.

The default settings in MATLAB for these tolerances are $\texttt{RelTol} = 10^{-3}$ and $\texttt{AbsTol} = 10^{-6}$.

The default `RelTol` is meant to ensure sufficient accuracy for plotting purposes.

`RelTol` $=10^{-5}$ is more typical for scientific computing.

`RelTol` $=10^{-10}$ should be about the lowest value you should reasonably be expected to use. For this purpose the solution you compute will be essentially "exact"[7]. In fact, in double precision, the minimum allowable value of `RelTol` is 2.22045×10^{-14}: obtaining more digits of accuracy is simply not feasible without going to higher precision floating-point arithmetic.

Solvers produce vectors $\mathbf{y}_n$ that approximate the solution $\mathbf{y}(t_n)$ on a mesh $0 = t_0 < \ldots < t_n = b$.

Roughly speaking, they aim to produce an approximation that satisfies

$$|y_i(t_n) - y_{n,i}| \leq \texttt{RelTol}|y_i(t_n)| + \texttt{AbsTol} \qquad (6)$$

for each component of the solution.

[7]assuming things go as they should!

An important rule of thumb

From Brennan, Campbell, and Petzold (1996):

> *We cannot emphasize strongly enough the importance of carefully selecting these tolerances to accurately reflect the scale of the problem.* In particular, for problems whose solution components are scaled very differently from each other, it is advisable to provide the code with vector-valued tolerances. For users who are not sure how to set the tolerances RTOL and ATOL, we recommend starting with the following rule of thumb. Let m be the number of significant digits required for solution component y_i. Set RTOL$_i = 10^{-(m+1)}$. Set ATOL$_i$ to be the value at which $|y_i|$ is essentially insignificant.

Inequality (6) defines a so-called *mixed* error control: if $\texttt{RelTol} = 0$, we have pure absolute error control; if $\texttt{AbsTol} = 0$, we have pure relative control.

A mixed strategy is generally employed because each pure strategy has difficulties associated with it.

Pure relative error control requires that

$$\left| \frac{y_i(t_n) - y_{n,i}}{y_i(t_n)} \right| \leq \texttt{RelTol}.$$

There are two serious problems with this:

1. $y_i(t_n)$ may vanish.

2. It is possible to ask for impossible accuracy (e.g., 16 digits in double precision) and a solver might not indicate any problems except to incur a *much* higher computational cost and possibly to produce answers that are more accurate at less stringent tolerances.

A pure absolute error control requires that

$$|y_i(t_n) - y_{n,i}| \leq \texttt{AbsTol}, \qquad i = 1, 2, \ldots, m.$$

Here the problem is that you need to judge how large the solution components will be, and you can get into trouble if you are off by too much.

Re-writing the pure absolute error control criterion as

$$\left|\frac{y_i(t_n) - y_{n,i}}{y_i(t_n)}\right| \leq \frac{\texttt{AbsTol}}{|y_i(t_n)|}, \qquad i = 1, 2, \ldots, m,$$

we see that a pure absolute error control of `AbsTol` on $y_i(t)$ corresponds to a pure relative error control of $\texttt{AbsTol}/|y_i(t_n)|$.

So if $|y_i(t_n)|$ is very large (e.g., 10^{11}), even an unremarkable `AbsTol` (e.g., 10^{-6}) may lead to an impossible accuracy request.

The other problematic situation for pure absolute error control occurs when $|y_i(t)| < \texttt{AbsTol}$.

In this case, *any* $y_{n,i}$ such that $|y_{n,i}| < \texttt{AbsTol}$ will pass the test, and an acceptable approximation will have *no* correct digits!

This would be fine if $y_i(t)$ was uninteresting at such low levels **and stayed there!** The danger is that it might grow later and become interesting again (then it would be hard to have much confidence in the solution).

Note 9. *Thankfully, you often have a correct digit or two in such cases even if you didn't expect it based on your choice of* `AbsTol`[8].

One reason for this is that the solver may have needed the accuracy on this small component to attain a specified accuracy on another component that depended on it.

Another reason is that the solver may have been forced to take a small step size for another component. This step size is generally smaller than necessary for other components, so they are computed more accurately than required.

[8] But this is not a free lunch!

Example: Proton transfer

This example of proton transfer in a $H_2 - H_2$ bond is taken from Lapidus, Aiken, and Liu (1973).

The phenomenon is described by the following ODEs:

$$\begin{aligned}
\dot{x}_1 &= -k_1 x_1 + k_2 y, \\
\dot{x}_2 &= -k_4 x_2 + k_3 y, \\
\dot{y} &= k_1 x_1 + k_4 x_2 - (k_2 + k_3) y,
\end{aligned}$$

subject to the initial conditions

$$x_1(0) = 0,\ x_2(0) = 1,\ y(0) = 0,$$

for $0 \le t \le 8 \times 10^5$. The constants are

$$\begin{aligned}
k_1 &= 8.4303270 \times 10^{-10}, & k_2 &= 2.9002673 \times 10^{11}, \\
k_3 &= 2.4603642 \times 10^{10}, & k_4 &= 8.7600580 \times 10^{-6}.
\end{aligned}$$

It turns out that this is a *stiff* problem.

See `LapidusAikenLiu.m`

It is not rare when modelling chemical reactions that concentrations below a certain threshold have negligible effects and so are uninteresting.

It makes sense to set `AbsTol` to reflect these thresholds.

Although theoretically these are positive quantities, if they are small enough, a solver may generate negative numbers.

These negative numbers are usually within `AbsTol`, and they decrease in magnitude with decreasing tolerances.

Note that this is completely permitted by the error control mechanism!

Sometimes this is nothing more than a small annoyance; other times a problem will not tolerate any negative values and become unstable.

How to have robust error control while maintaining non-negativity of solution components is an open question.

Robertson's problem (1966)

This is a popular benchmark problem for stiff IVP solvers. It describes a chemical reaction governed by the IVP

$$\begin{aligned}
\dot{y}_1 &= -0.04 y_1 + 10^4 y_2 y_3, & y_1(0) &= 1,\\
\dot{y}_2 &= 0.04 y_1 - 10^4 y_2 y_3 - 3\times 10^7 y_2^2, & y_2(0) &= 0,\\
\dot{y}_3 &= 3\times 10^7 y_2^2, & y_3(0) &= 0,
\end{aligned}$$

on the interval $0 \le t \le 4\times 10^6$.

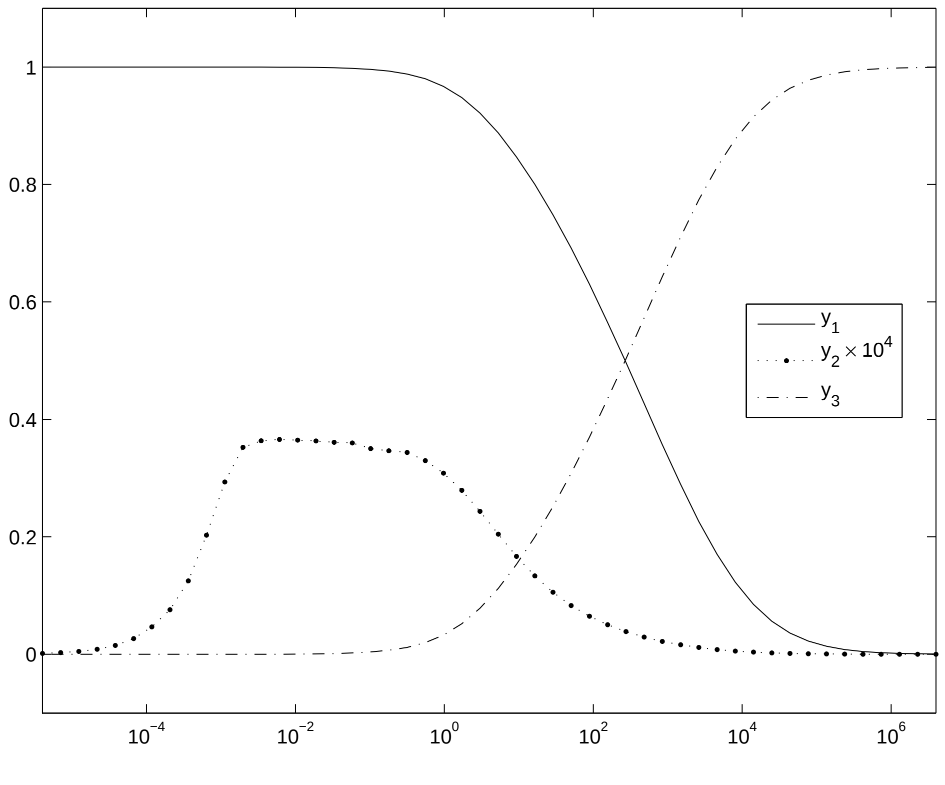
It is easy to show that $y_i(t) \ge 0$, $i = 1, 2, 3$ and

$$y_1(t) + y_2(t) + y_3(t) \equiv 1$$

for all $t \ge 0$.

This problem is included in MATLAB as the demo program `hb1ode`.

Here is what the solution (produced by `ode15s`) looks like:



Hindmarsh and Byrne (1976) use this problem to show the performance of their code `EPISODE` for stiff IVPs.

With `AbsTol` $= 10^{-6}$, a small negative component emerges. It grows rapidly in magnitude, and soon the numerical solution becomes unacceptable; see table.

t	y_1	y_2	y_3
4e5	4.9394e−03	1.9854e−08	9.9506e−01
4e7	3.2146e−05	1.2859e−10	9.9997e−01
4e9	–1.8616e+06	–4.0000e−06	1.8616e+06

Table 1: Robertson's problem; steady-state solution computed using `EPISODE`.

We emphasize here that the unsatisfactory performance is a consequence of the problem and what is asked of the solver!

Other solvers exhibit similar problems operating in this way (including `ode15s`).

Accuracy: Too much vs. too little

We have seen that you can ask for too much relative accuracy.

However, asking for too little accuracy can give you other headaches!

It is tempting to ask for as little accuracy as you can get away with because computational expense increases with increased requested accuracy.

Moreover often data of a problem are only known to a digit or two.

However, *it is dangerous and pointless to ask for too little accuracy!*

Solvers are based on theory that is only valid when step sizes are sufficiently small. If you ask for little accuracy, a solver may take steps that are too large for reliable results.

Recall: solvers only directly control local errors; they control global errors indirectly. The local errors are made to be smaller than the requested tolerances so that in the end the global errors will be smaller than (or comparable to!) the tolerances.

If an IVP is unstable or its solution is highly oscillatory, then asking for too little accuracy leads to grossly inaccurate answers: the solution may be physically unrealistic, (or worse) the solution may be plausible but incorrect, or the computation may fail entirely.

If the solver is reputable, unsatisfactory results are usually a consequence of IVP instability; other solvers would (at best!) exhibit similar behaviour under the same circumstances.

A general plan

In general you should never solve an IVP once and think that you have your final answer.

You can expect to have to solve it several times, especially if the problem is new to you, until you find an appropriate range for the tolerances.

A general plan is to solve a sequence of problems with decreasing tolerances to ensure you have achieved the accuracy you desire.

1.5 Qualitative Properties

We have seen problems with solutions that have certain qualitative properties that are implied by the ODEs.

Generally speaking, numerical solutions will **not** inherit these properties.

Although there are specialized methods that do preserve certain qualitative solution properties, we do not consider them here.

The important exception to this is that standard numerical methods **do** preserve *linear invariants*.

Consider the standard IVP:

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(0) = \mathbf{y}_0.$$

If there exists a constant, non-zero vector $\mathbf{c}$ such that $\mathbf{c}^T\mathbf{f}(t, \mathbf{y}) = 0$ for all $(t, \mathbf{y}(t))$, then $\mathbf{y}(t)$ satisfies the *linear invariant*[9]

$$\mathbf{c}^T\mathbf{y}(t) \equiv \mathbf{c}^T\mathbf{y}_0 = \text{constant.}$$

Linear invariants express physical laws such as conservation of mass or charge.

We have seen that the $H_2 - H_2$ bond problem and Robertson's problem possessed linear invariants. (The sum of the components was always equal to 1.)

It turns out (see Shampine, 1998) that all standard numerical methods for IVPs preserve linear invariants (to within roundoff errors).

Just because linear invariants are satisfied does not mean that a numerical solution is any good! (See results from `EPSIODE`.)

[9] also called a *linear conservation law*

However, if a standard method does not satisfy linear invariants (to within roundoff errors), then either there is a bug in the implementation or the rounding errors in the computation are significant.

What other kinds of invariants are there?

For example, the frictionless pendulum system preserves energy.

A general numerical solution to this problem will have an energy that is only approximately constant:

Suppose the solver at time t_n produces approximations

$$y_{n,1} = \theta(t_n) + e_1, \quad y_{n,2} = \dot{\theta}(t_n) + e_2,$$

where e_1, e_2 are small errors.

Given that the energy of the pendulum system is

$$E(\theta, \dot{\theta}) = \frac{1}{2}\dot{\theta}^2 - \cos\theta,$$

the energy of the numerical solution is approximately

$$E(y_{n,1}, y_{n,2}) \approx E(\theta(t_n), \dot{\theta}(t_n)) + \dot{\theta}(t_n)e_2 + \sin\theta(t_n)e_1.$$

So the error in the energy is comparable to the errors in the components; hence $E \approx$ constant.

This is often OK.

However, long-term qualitative behaviour of the solution may depend critically on this difference.

Of course you can make this difference smaller by asking for more accuracy, but this can be prohibitively expensive for long-time simulations.

If conservation of a *nonlinear invariant* (such as energy for the pendulum) is critical, then it is best to search for a specialized numerical method that will do this.

But again, there is no free lunch!

Satisfying the nonlinear invariant without incurring exorbitant computational costs does come at the expense of (so-called pointwise) accuracy.

For example, in the case of the pendulum, you might get the proper height of the orbit, but you would get the wrong period of motion!

So after a while, the position of the pendulum would not be approximated well.

As another example, the 2-body problem satisfies two nonlinear invariants:

1. Conservation of energy

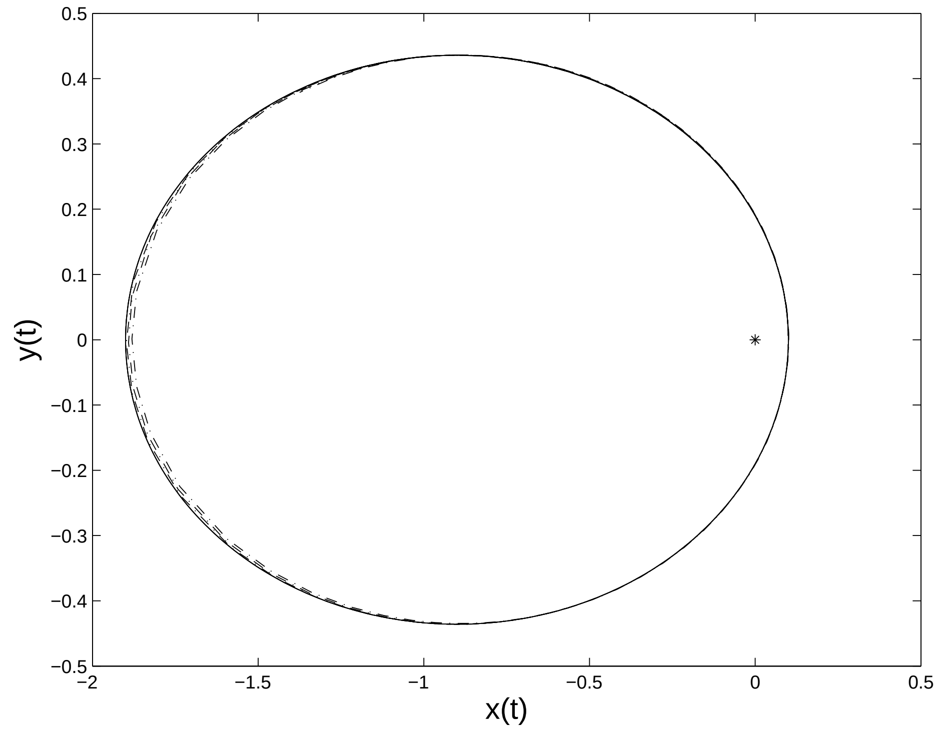
$$E = \frac{\dot{x}^2(t) + \dot{y(t)}^2}{2} - \frac{1}{r(t)},$$

   where $r(t) = \sqrt{x^2(t) + y^2(t)}$.

2. Conservation of angular momentum

$$L = x(t)\dot{y}(t) - y(t)\dot{x}(t).$$

Here is a picture of some typical behaviour:



With tight tolerances, the orbit is closed; energy and angular momentum are constant to within plotting accuracy.

With the default tolerances, `ode15s` removes energy from the system; the satellite falls into the fixed body. The same is true for `ode113`. On the other hand, `ode45` adds energy to the system, and the satellite spirals away from the fixed body.

Note 10. *Some energy is lost when using* `ode15s` *even for tight tolerances! i.e., the satellite will fall into the fixed body if the interval of integration is sufficiently long.*

(Analogous comments can be made for `ode113` *and* `ode45`*.)*

This is a qualitatively different behaviour of the system to that which is correct!

If a nonlinear invariant like energy conservation is important for this reason, you will have to turn to more exotic numerical methods such as *symplectic* or *variational* integrators.

The overall shape of the orbit will be preserved, but, as mentioned, the exact position on the orbit for a given time will be lost.

The only way to have your cake and eat it too is to pay for it.

In other words, you cannot have it both ways on a fixed budget unless that budget is sufficiently large.