Research Computing with Python, Lecture 7, Numerical Integration and Solving Ordinary Differential Equations

Ramses van Zon

SciNet HPC Consortium

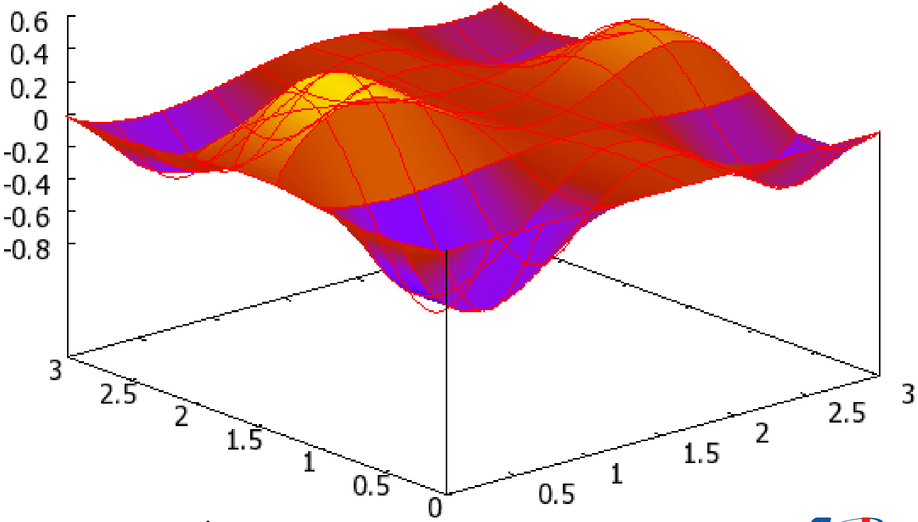
November 26, 2013

Today's Lecture

- Numerical Integration
- Ordinary Differential Equations
- Little bit of theory
- How to do this in Python
  (spoiler: use `scipy.integrate`)

Numerical Integration

Numerical Integration

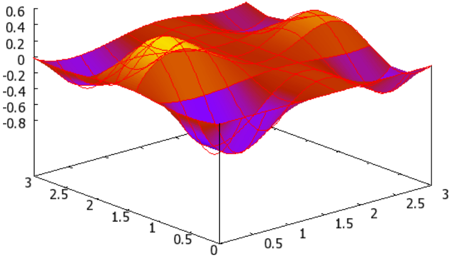


$\mathcal{I} = \int_{\mathcal{D}} \mathbf{f}(\mathbf{x})\, \mathbf{d}^{\mathbf{d}}\mathbf{x}$

Numerical Integration Methods

If our integral cannot be computed exactly, what options do we have?

.



Method depends on dimension d, function f(x), and x-domain.

.

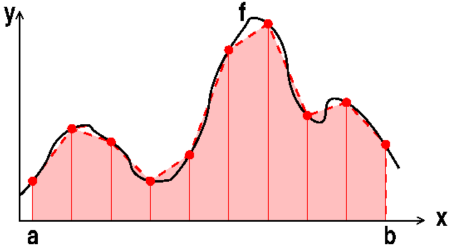
d=1:	• Regular grid • Gaussian Quadrature
d small:	• Regular grid • Recursive Quadrature
$d \gg 1$:	• Monte Carlo

.

Regularly spaced grid methods

Problem:

- A curve is given by an function y=f(x).
- The area under the curve is required, between a and b.



Numerical approach:

- Compute the value of y at equally space points x
- Using an interpolation function between those points, compute area

In the figure:

- Linear interpolation: *trapezoidal rule*
- The shaded area is returned by this approach
- This is an approximation to the actual area.

Equally spaced grid approach

- Compute the value of y at equally space points x
- Trapezoidal rule:

$$\mathcal{I} = \frac{1}{2}y_1 + \sum_{i=2}^{n-1} y_i + \frac{1}{2}y_n$$

```
def f(x):
 return cos(x/9)*sin(x)**2
a=0
b=10
x=linspace(a,b,40)
dx=x[1]-x[0]
y=f(x)*dx
I1=(y[0]+y[-1])/2+sum(y[1:-1]
print I1
3.93845493792
```

Different evenly spaced grid approaches

- Trapezoidal

$$\int_a^{a+h} f(x)\,dx \approx \frac{h}{2}\left[f(a) + f(a+h)\right]$$

- Simpson

$$\int_a^{a+2h} f(x)\,dx \approx h\left[\frac{1}{3}f(a) + \frac{4}{3}f(a+\frac{h}{2}) + \frac{1}{3}f(a+h)\right]$$

- Bode, Backward differentiation, ...
- Different prefactors, different orders, different points

What you use is the extension of these rules to multiple intervals.

Unevenly spaced grids

Gaussian quadrature

- Based on orthogonal polynomials on the interval.
- E.g. Legendre, Chebyshev, Hermite, Jacobi polynomials
- Compute and $\mathbf{y_i = f(x_i)}$ then

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} v_i f_i$$

  $\mathbf{x_i}$ and $\mathbf{v_i}$ from polynomial properties
- Tend to be more accurate than equally spaced approaches

```
# nth order Gauss-Legendre quadrature:
from scipy.integrate import fixed_quad
I2=fixed_quad(f,a,b,n=20)[0]
print I2
3.9363858769075524
```

Accuracy

Was this the right value?

- Always an approximation
- More points means better approxiation
- If curve is smooth, better interpolation means better approximation (why unevenly spaced points helps)
- But how close are we?

Adaptive Integration

Rather than choosing a 'safe' large number of n, we should increase number of points until a *given accuracy* is achieved

Adaptive Integration

```
#Adaptive Gauss-Legendre integration
from scipy.integrate import quad
I3=quad(f,a,b,epsrel=0.001)
print I3
(3.936385876907544, 0.0009622632189420763)
```

Arguments of interest for quad

- f: The function
- a,b: The x limits
- epsabs: Absolute error tolerance.
- epsrel: Relative error tolerance.
- limit : An upper bound on the number of subintervals used in the adaptive algorithm.

Numerical Integration in d > 1 but small

Why multidimensional integration is hard:

- Requires $\mathcal{O}(\mathbf{n^d})$ points if its 1d counterpart requires $\mathbf{n}$.
- A function can be peaked, and peak can easily be missed.
- The domain itself can be complicated.

Numerical Integration in d > 1 but small

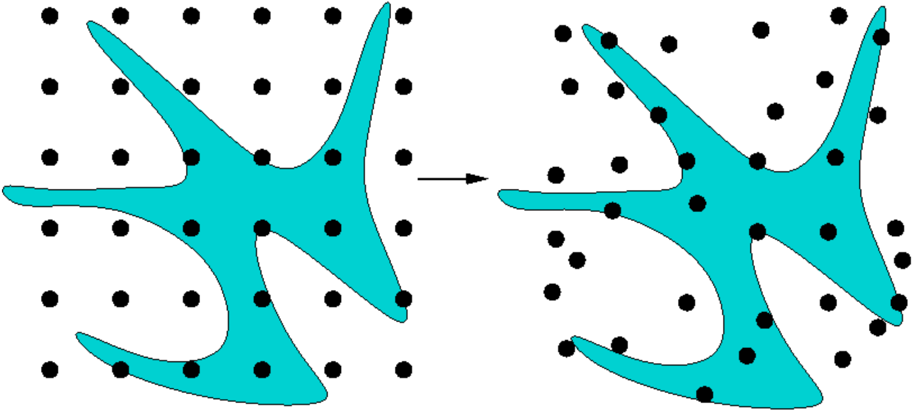
So what should you do?

- If you can reduce the d by exploiting symmetry or doing part of the integral analytically, do it!
- If you know the function to integrate is smooth and its domain is fairly simple, you could do repeated 1d integrals (fixed-grid or Gaussian quadrature)
- Otherwise, you'll have to consider Monte Carlo.

```
from scipy.integrate import d
def f(x,y):
    return x*y
def y1(x):
    return 0
def y2(x):
    return 3.14
a=0; b=3.14
I4=dblquad(f,a,b,y1,y2)
print I4[0]
24.30292804
```

Monte Carlo Integration

Use random numbers to pick points at which to evaluate integrand.



- Convergence always as $\mathbf{1}/\sqrt{\mathbf{n}}$, regardless of $\mathbf{d}$.
- Simple and flexible.

Monte Carlo Integration

- You can find python packages for MC (not in scipy, though)
- But the essence is the same:
  1. Use random numbers to generate points in your domain
  2. Evaluate the function on those points
  3. Average them and compute standard deviation for error.
- One variation is to use a bias in step 1 to focus on regions of interest. Bias can be undone in averaging step
- Another variation is to have each point generated from the previous one plus a random component: MC chain.

Ordinary Differential Equations

Ordinary Differential Equations (ODE)

Lotka–Volterra

$$\frac{dx}{dt} = x(\alpha - \beta y)$$

$$\frac{dy}{dt} = -y(\gamma - \delta x)$$

Harmonic oscillator

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = -x$$

Rate equations

$$\frac{dx}{dt} = -2k_1x^2y + 2k_2z^2$$

$$\frac{dy}{dt} = -k_1x^2y + k_2z^2$$

$$\frac{dz}{dt} = 2k_1x^2y - 2k_2z^2$$

Lorenz system

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

Mathematical Details

- General form:

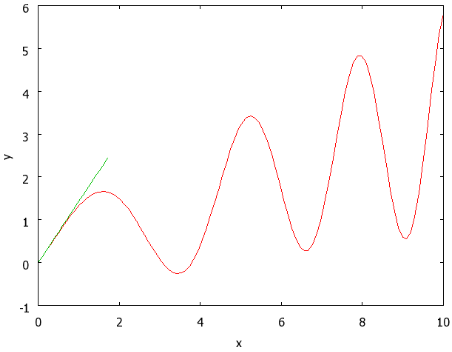
$$\sum_n a_n(t, y) \frac{d^n y}{dt^n} = f(t, y)$$

  $\mathbf{n}$=order

- Boundary conditions: much like PDEs: next lecture

- **Initial conditions**: $\mathbf{y}, \frac{dy}{dx}, \ldots,$ at $\mathbf{t = t_0}$

- Define $\mathbf{y_0 = y}$; $\mathbf{y_1 = \frac{dy}{dx}}, \ldots,$ $\rightarrow$ set of first order ODEs



First order initial value problem

- Start from the general first order form:

$$\frac{dy}{dt} = f(t, y)$$

- t is one dimensional, y can have multiple components
- All approaches will evaluate f at discrete points $t_0, t_1, \ldots$.
- Like integration:

$$y_{n+1} = y_n + \int_t^{t+h} f(t', y(t'))dt'$$

- Consecutive points may have a fixed step size $h = x_{k+1} - x_k$ or may be adaptive.
- $y_j(t_{n+1})$ may be implicitly dependent on $f(t_{nr+1})$.

Stiff ODEs

- A stiff ODE is one that is hard to solve, i.e. requiring a very small stepsize **h** or leading to instabilities in some algoritms.
- Usually due to wide variation of time scales in the ODEs.
- Not all methods equally suited for stiff ODEs. Implicit ones tend to be better for stiff problems.

ODE solver algorithms: Euler

To solve:

$$\frac{dy}{dt} = f(t, y)$$

Simple approximation:

$$y_{n+1} \approx y_n + hf(t_n, y_n) \qquad \text{“forward Euler”}$$

Rationale:

$$y(t_n + h) = y(t_n) + h\frac{dy}{dt}(t_n) + \mathcal{O}(h^2)$$

So:

$$y(t_n + h) = y(t_n) + hf(t_n, y_n) + \mathcal{O}(h^2)$$

- $\mathcal{O}(h^2)$ is the local error.
- For given interval $[t_1, t_2]$, there are $n = (t_2 - t_1)/h$ steps
- Global error: $n \times \mathcal{O}(h^2) = \mathcal{O}(h)$
- Not very accurate, nor very stable (next): don't use.

Stability

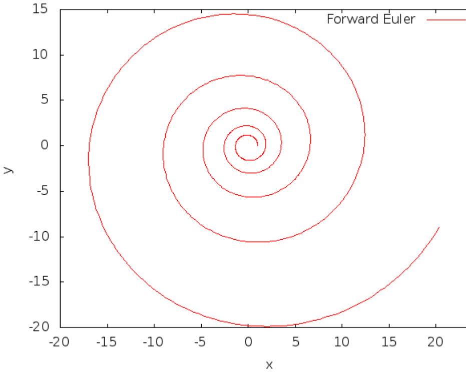
Example: solve harmonic oscillator numerically:

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = -x$$

Using Euler gives

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$



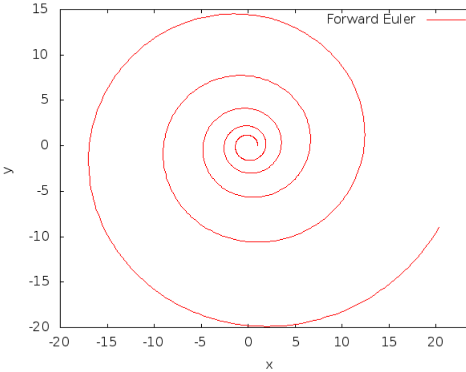
Stability

Example: solve harmonic oscillator numerically:

$$\frac{dx}{dt} = y$$
$$\frac{dy}{dt} = -x$$

Using Euler gives

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$



Stability: eigenvalues $\lambda_\pm = 1 \pm ih$ of that matrix.
$|\lambda_\pm| = \sqrt{1+h^2} > 1 \quad \Rightarrow$ Unstable for any **h**!

ODE algorithms: implicit mid-point Euler

To solve:

$$\frac{dy}{dt} = f(t, y)$$

Symmetric simple approximation:

$$y_{n+1} \approx y_n + hf(x_n, (y_n + y_{n+1})/2) \qquad \text{“mid-point Euler”}$$

This is an implicit formula, i.e., has to be solved for y_{n+1}.

Example: Harmonic oscillator

$$\begin{pmatrix} 1 & -\frac{h}{2} \\ \frac{h}{2} & 1 \end{pmatrix} \begin{pmatrix} y^{[1]}_{n+1} \\ y^{[2]}_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \frac{h}{2} \\ -\frac{h}{2} & 1 \end{pmatrix} \begin{pmatrix} y^{[1]}_{n} \\ y^{[2]}_{n} \end{pmatrix}$$

Eigenvalues $\mathbf{M}$ are $\lambda\pm = \frac{(1\pm ih/2)^2}{1+h^2/4}$ so $|\lambda_\pm| = 1 \Rightarrow$ **Stable**

ODE solver algorithms: Predictor-Corrector

- Computation of new point
- Correction using that new point
- Gear P.C.: keep previous values of **y** to do higher order Taylor series (predictor), then use **f** in last point to correct. Can suffer from catestrophic cancellation at very low **h**.
- Adams: Similarly uses past points to compute.
- Runge-Kutta: Refines by using mid-points.
- Some schemes require correction until convergence.
- Some schemes can use the *jacobian*, e.g. the derivatives of the right hand side.

Further ODE solver techniques

Adaptive methods:

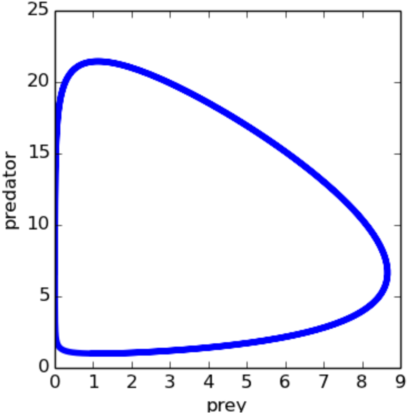
As with the integration, rather than taking a fixed **h**, vary **h** such that the solution has a certain accuracy.

- Don't code this yourself!
- Good schemes are implemented in packages such as `scipy.integrate.odeint`, `scipy.integrate.ode`
- `odeint` uses an Adams integrator for non-stiff problems, and a backwards differentiation method for stiff problem.
- ode is a bit more flexible.

Lotka–Volterra using scipy.integrate.odeint

$$\frac{dx}{dt} = x(\alpha - \beta y)$$

$$\frac{dy}{dt} = -y(\gamma - \delta x)$$




```
from scipy.integrate\
    import odeint
alpha=0.1
beta=0.015
gamma=0.0225
delta=0.02
def system(z,t):
    x,y=z[0],z[1]
    dxdt= x*(alpha-beta*y)
    dydt=-y*(gamma-delta*x)
    return [dxdt,dydt]
t=linspace(0,300.,1000)
x0,y0=1.0,1.0
sol=odeint(system,[x0,y0],t)
X,Y=sol[:,0],sol[:,1]
plot(X,Y)
```

Conclusions

Conclusions

- Many different methods for numerical integration and solving ODEs
- Python package `scipy.integrate` helps you out.
- It has procedures to readily get good results: `scipy.integrate.quad` and `scipy.integrate.odeint`
- Unfortunately, hard to get what they really do:
  - For `scipy.integrate.quad`, had to look into the scipy python source to know that it uses Legendre polynomials.
  - For `scipy.integrate.odeint`, had to look into the fortran documentation
- If you're using sciPy for anything but exploration: do you research and learn what they really do!

Next Time

Next and Final Lecture

Thursday November 28, 2013, 11:00 am
Topic: Partial differential equations