Laboration: Ordinära differentialekvationer

1 Kort introduktion

Ordinära differentialekvationer (ODE:er) kan användas för att beskriva en stor mängd fysikaliska fenomen. Att kunna hantera ODE:er är också mycket relevant vid lösning av partiella differentialekvationer. Likt integraler och icke-linjära ekvationer så är ODE:er ofta svåra eller omöjliga att lösa analytiskt, då används i stället numeriska metoder.

- Man kan dela in ODE:er i två huvudtyper: begynnelsevärdesproblem och randvärdesproblem. I den här laborationen kommer vi enbart titta på begynnelsevärdesproblem. Ett sådant problem kan skrivas

$$\begin{aligned} y'(t) &= f(t, y), \quad t > a \\ y(a) &= y_0, \end{aligned} \tag{1}$$

  där a och y_0 är kända parametrar och $f(t, y)$ en känd funktion.

- Grundprincipen när ODE:er löses numeriskt är att det intervall man vill lösa problemet på delas in i punkter (diskretiseras) och sedan approximeras lösningen enligt någon princip i dessa punkter. Avståndet ifrån en punkt till nästa kallas steglängd. Detta liknar diskretiseringen vi gjorde vid numerisk integrering av en funktion på ett intervall.

- Till de numeriska metoderna som används för att lösa ODE:er är kopplat en så kallad noggrannhetsordning (likt metoderna för beräkning av integraler). Om en viss metod är effektiv och ”bra” avgörs bland annat (men inte enbart) av detta begrepp. Det är därför viktigt att förstå vad begreppet står för. Vi kommer i laborationen att empiriskt undersöka och åskådliggöra begreppet för tre olika numeriska metoder: Eulers metod, Heuns metod och klassisk Runge-Kutta (ibland kallad Runge-Kutta 4 eller RK4).

- Ett annat viktigt koncept är stabilitet. I laborationen kommer vi att undersöka begreppet numerisk stabilitet och hur det är kopplat till så kallade styva differentialekvationer. Du kommer också att få se skillnaden i hur explicita respektive implicita metoder fungerar för den typen av problem.

Börja med att ladda ner filerna för denna laboration: `Eulerdemo.py`, `accuracydemo.py` och `stabdemo.py`.

Obs! Om du använder en IDE så kan du behöva ställa in så att interaktiva figurer kan skapas. Till exempel i Spyder: Preferences → IPython Console → Graphics → Backend → Automatic.

2 Eulers metod

Du ska nu med hjälp av ett demoprogram titta på hur Eulers metod fungerar för den icke-linjära ODE:n

$$\begin{aligned} y'(t) &= \cos(yt), \quad 0 \leq t \leq 2\pi, \\ y(0) &= 0. \end{aligned} \tag{2}$$

Andra namn för samma metod är explicit Euler och Euler framåt. Metoden startar med initialvärdet vid $t = 0$ och stegar sig sedan framåt i tiden en steglängd i taget.

1. Kör programmet `Eulerdemo` med steglängderna 0.4, 0.2 och 0.1. Var uppmärksam på steglängdens inverkan på resultatet och noggrannheten.

3 Noggrannhet

Det finns många andra och bättre metoder för att lösa ODE:er än Eulers metod, men alla bygger på liknande grundläggande principer. Du ska nu studera noggrannhetsordningen för Eulers metod, Heuns metod och klassisk Runge-Kutta.

2. Kör programmet `accuracydemo` och försök förstå den figur som ritas. Programmet löser en linjär ODE och plottar felet som funktion av steglängden för Eulers metod, Heuns metod och klassisk Runge-Kutta. Lutningen av varje kurva motsvarar metodernas noggrannhetsordning, läs den printade texten. Notera att axlarna har en logaritmisk skalning. Det tycks finnas en gräns för hur noggrann lösning man kan beräkna, fundera på varför. Ungefär vid vilket fel finns denna gräns?

3. När man löser ODE:er vill man att metoden ska lösa problemet med en viss noggrannhet, ett visst fel. Här ska du titta på hur detta är kopplat till noggrannhetsordningen. I figuren från föregående uppgift (alltså programmet `accuracydemo`), välj en feltolerans och tryck `Get statistics`. Börja med feltoleransen 10^{-4} (kan skrivas `0.0001` eller `1e-4`) och studera informationen som printas. Programmet uppskattar den steglängd som krävs för att uppnå den givna toleransen och beräknar hur många steg och vilken körtid det motsvarar för varje metod. Var uppmärksam på skillnaderna i steglängd och antal beräkningssteg för de olika metoderna. Var också uppmärksam på att metoderna tar olika lång tid per beräkningssteg.

4 Stabilitet

För att undersöka stabiliteten hos en metod används ofta differentialekvationen

$$\begin{aligned} y'(t) &= \lambda y, \quad t \geq 0, \\ y(0) &= y_0, \end{aligned} \tag{3}$$

där λ är en negativ konstant, denna ekvation kallas ibland för *testekvationen*. Den exakta lösningen till (3) är $y(t) = y_0 e^{\lambda t}$. Vi ska nu undersöka stabiliteten för Eulers metod (explicit Euler) och Eulers implicita metod, vilken vanligtvis kallas implicit Euler eller Euler bakåt. Båda metoderna har samma noggrannhetsordning, men de skiljer sig åt när det kommer till stabilitet. För explicit Euler kan man teoretiskt visa att det finns en övre gräns för hur stor steglängd h som kan väljas, ett så kallat stabilitetsvillkor. För ODE:n (3) är detta villkor $h < 2/|\lambda|$. För implicit Euler finns ingen begränsning på steglängden för att uppnå en stabil lösning, dvs. den är ovillkorligt stabil.

4. Kör programmet `stabdemo`. Sätt steglängden $h = 0.05$ och $\lambda = -10$ och jämför metoderna. Ändra λ till exempelvis -100. Studera vilken effekt det får på lösningarna.

5. Undersök hur lösningen ser ut med explicit Euler om steglängden väljs till precis under, precis över och exakt på stabilitetsgränsen. Använd till exempel $\lambda = -20$ och steglängderna 0.09, 0.1 och 0.11. Med detta λ så är stabilitetsvillkoret $h < 0.1$. Försök förstå utifrån plottarna vad som menas med begreppet stabilitet? Vad händer med lösningen för de olika valen av steglängd?

5 SciPy-funktioner för ordinära differentialekvationer

För lösning av ODE:er i Python kan SciPy-funktionen `solve_ivp` användas. I funktionen finns ett antal metoder för begynnelsevärdesproblem (på engelska *initial value problems* eller IVP) implementerade, vilken metod som används bestämms av argumenten till funktionen. Vi ska här använda metoden RK45. Denna metod är en *adaptiv* Runge-Kutta metod, vilket betyder att steglängden i varje steg väljs så att en viss noggrannhet erhålls. På så sätt anpassas steglängden till hur funktionen ser ut, likt adaptiva metoder för beräkning av integraler. Funktionen anropas enligt följande:

```
1  import scipy.integrate as ode
2  SOL = ode.solve_ivp(fun,tspan,y0,method='RK45',args=args)
```

Här är `fun` en Python-funktion `fun(t,y,...)` som definierar högerledet av ODE:n, `tspan` en tuple med start- och stopptid och `y0` initialdatan. Argumentet `args` används för att skicka in ytterligare parametrar till högerledsfunktionen. Lösningen och övrig användbar information lagras i variabeln `SOL`. För ytterligare information och exempel på hur `solve_ivp` används, se SciPy-dokumentationen på `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`.

Som exempelproblem ska vi studera en matematisk modell för epidemier. Beteckna antalet mottagliga för en sjukdom vid tidpunkten t för $S(t)$ ("susceptible"), antalet infekterade ("infectious") som kan sprida smittan för $I(t)$ och antalet immuna ("recovered") för $R(t)$. Modellen ges av ODE-systemet

$$\begin{aligned} S'(t) &= bN - dS - \beta\frac{I}{N}S - vS, \\ I'(t) &= \beta\frac{I}{N}S - uI - dI, \\ R'(t) &= uI - dR + vS, \end{aligned} \tag{4}$$

där N är storleken på populationen, b motsvarar andel inflyttade och nyfödda, d är andel döda eller utflyttade, β är antal kontakter som en individ har per tidsenhet som orsakar smitta, u är andel sjuka som blir immuna per tidsenhet och v är andel mottagliga som blir vaccinerade per tidsenhet.

Uppgiften är att simulera en epidemi från tiden $t = 0$ till 60 dagar med hjälp av SciPy-funktionen `solve_ivp`. Låt $\beta = 0.3$, $u = 1/7$, vilket motsvarar en infektionstid på 7 dagar, och $v = 0$, dvs. ingen vaccination. Med $b = 0.002/365$ och $d = 0.0016/365$ erhålls ungefär samma andel födda och döda per dag som i Sverige. Antag att vid tidpunkten $t = 0$ är 5 personer infekterade och ingen är immun. Notera att $S(t) + I(t) + R(t) = N$, dvs. samtliga i populationen tillhör alltid en av grupperna. Låt $N = 1000$. Skriv programmet så att du enkelt kan ändra alla parametrar på ett ställe.

6. Börja med att definiera högerledsfunktionen. Funktionshuvudet ska vara:

   ```
   1  def f(t,y,N,b,d,beta,u,v):
   ```

   I det här fallet ska `y` vara en Numpy-array med tre element motsvarande S, I och R och funktionen ska returnera en Numpy-array med tre element motsvarande S', I' och R'. Notera att den oberoende variabeln, i det här fallet t, måste vara med som inparameter även fast den inte används i högerledsfunktionen.

7. Lös ODE:n med hjälp av `solve_ivp` och metoden RK45. Plotta lösningen, dvs. plotta $S(t)$, $I(t)$ och $R(t)$ som en funktion av t. Ser resultatet rimligt ut?

8. När du fått ditt program att fungera kan du testa att lägga in ett vaccinationsprogram, tex. genom att sätta $v = 0.01$. Variera även andra parametrar och undersök vad som händer. Du kan till exempel minska sjukdomstiden till 3 dagar, dvs. sätt $u = 1/3$, och öka eller minska β.

Tips: Med argumentet `max_step` kan man sätta en övre gräns på hur stora steg den adaptiva metoden får ta. Lägg till `max_step = 1` i funktionsanropet av `solve_ivp` för att garantera att tillståndet beräknas minst en gång om dagen. På så sätt fås en mjukare plot.