Datorlaborationer, Datorer och datoranvändning

Datorlaborationerna ger exempel på tillämpningar av det material som behandlas under kursen. Schema för laborationerna (tider och datorsalar) finns i kursprogrammet och på kurshemsidan. För laborationerna gäller följande:

- Laborationerna är obligatoriska. Det betyder att du måste bli godkänd på alla uppgifterna under ordinarie laborationstid. Om du skulle vara sjuk vid något laborationstillfälle så måste du anmäla detta till kursansvarig (`mattias.nordahl@cs.lth.se`) före laborationen. Om du varit sjuk bör du göra uppgiften på egen hand och redovisa den under ditt nästa laborationstillfälle. Det kommer också att anordnas en uppsamlingslaboration efter kursens slut, för de studenter som missat någon laboration.
- Uppgifterna i laborationerna ska lösas individuellt. Regler för samarbete finns på nästa sida.
- Varje laboration består av två delar: hemarbete och datorarbete. Innan du kommer till laborationen ska du ha förberett dig genom att ha gjort uppgifterna som markerats som hemarbete, samt läst igenom vad som står under datorarbete. Du ska också ha gått igenom kontrollfrågorna.
- Om du hittar någonting i uppgifterna eller andra anvisningar som är felaktigt eller oklart så är vi tacksamma om du meddelar dina synpunkter till `mattias.nordahl@cs.lth.se`.
- I början av varje laboration kommer laborationsledaren att kontrollera att du har förberett dig. Kontrollen görs genom att du får en skrivning med fyra frågor som valts bland kontrollfrågorna. Du måste besvara minst tre av frågorna korrekt för att du ska få genomföra laborationen.
- Observera att laborationerna inte syftar till att testa er, utan är ett inlärningsmoment! Under laborationerna får ni ta hjälp av allt kursmaterial eller andra resurser för att lösa uppgifterna.
- Svaren till kontrollfrågorna kommer att finnas i laborationshandledningen eller i annat material som delas ut eller hänvisas till. Du kommer också behöva tänka eller reflektera själv med utgångspunkt från materialet.

Riktlinjer för inlämningsuppgifter och laborationsuppgifter

Bland LTHs gemensamma regler finns följande föreskrifter:

- Inlämningsuppgifter skall fullgöras individuellt om det inte särskilt anges att de skall fullgöras i grupp.
- Vid arbete i grupp bestämmer ansvarig lärare om gruppindelningen och ändringar av denna. Arbetet skall utföras av dem som ingår i gruppen.
- Det är tillåtet att diskutera uppgifterna och tolkningen av dessa med utomstående på ett allmänt plan men inte att få hjälp med de konkreta lösningarna.
- Det är inte tillåtet att kopiera annans eller annan grupps lösningar helt eller delvis. Det är inte heller tillåtet att kopiera från exempelvis litteratur eller Internet. Vid citat skall källan tydligt anges.
- Väsentlig hjälp, av annan än lärare på kursen, för att genomföra en uppgift skall redovisas i redogörelsen eller på annat tydligt sätt. Detsamma gäller om man använt någon annan form av hjälpmedel som läraren inte kan förutsättas känna till.
- Institutionerna kan komplettera dessa regler skriftligen i samband med kursstarten, exempelvis i ett kursprogram.

Vid institutionen för datavetenskap gäller även följande kompletteringar/förtydliganden av reglerna:

- Reglerna om arbete i grupp ovan tillämpas för alla obligatoriska moment som utförs i grupp, det vill säga även laborationer och projektarbeten.
- Då arbete görs i grupp skall alla gruppdeltagare delta i arbetet.
- Hjälp från annan med handhavandet av apparatur, utnyttjandet av datorsystem och givna datorprogram behöver inte redovisas.

Kontakta ansvarig lärare om du är osäker på om viss hjälp är tillåten eller inte!

Institutionen tillämpar dessa riktlinjer på alla kurser. Om vi är övertygade om att fusk skett så överlämnar vi ärendet till universitetets disciplinnämnd för vidare åtgärd. Finner disciplinnämnden de studerande skyldiga är påföljden upp till sex månaders avstängning från universitetet och högskolan.

Tips

Här samlas kort information och hjälp som kan vara användbar under laborationerna. Vid problem och frågor under laborationerna, kolla gärna om din fråga finns besvarad här.

- `Vad menas med att 'gå till /usr/local/cs/dod/…'?`
  Det är en absolut sökväg som finns tillgänglig när du är inloggad på skolans datorer. (ILL 1.7)

- `Om jag använder min egen laptop, hur kommer jag åt /usr/local/cs/dod/…`?
  Du kan använda `ssh` för att logga in och arbeta på skoldatorerna (ILL 3.7), eller kopiera filer från skoldatorerna till din egen med `scp` eller `sftp` (ILL 4.2).

- `Hur stänger jag vim!?`
  Om du råkat öppna texteditorn `vim` (t.ex. genom att göra en git commit utan att ha ställt in en annan editor) kan du stänga den genom att trycka `:q` (kolon följt av `q`), sedan `Retur`. Om du redan provat annat kan du ha kommit in i något annat av Vim's editeringslägen. Tryck då först `Esc` för att gå tillbaka till det "normala läget".

- `Hur arbetar man med texteditorn nano?`
  nano är en terminalbaserat texteditor. Den körs alltså direkt i terminalen utan att öppna några nya fönster, vilket kan vara fördelaktigt ibland. I editorn kan du flytta markören med piltangenterna och skriva text som förväntat. I botten av terminalen visas också operationer som kan utföras och vilken knappkombination som ska tryckas. Där används tecknet `^` (den lilla "hatten" vid sidan om RETURN) för att betyda Ctrl-knappen. Använd t.ex. CONTROL-O för att spara (Write Out), CONTROL-X för att avsluta eller CONTROL-G för mer hjälp.

Laboration 1 — Linux/Unix

Mål: Du ska bekanta dig med att använda LTHs Linuxdatorer. Du blir inte expert på Linux (eller Unix) på en laboration, och det behöver du inte heller vara. Men det är viktigt att du är van att arbeta med Linux (och därmed även Unix); det kommer att underlätta dina studier i fortsättningen.

Hemarbete

H1. Läs igenom *Appendix B Terminalfönster* i *Introduktion till programmering med Scala* av Björn Regnell.

H2. Läs igenom kompendiet *Introduktion till LTH:s Linuxdatorer*. Kompendiet är tämligen långt, så börja i god tid. Laborationen innefattar ungefär det som finns i kapitel 1 (Grunderna) och 2 (Påbyggnad). De resterande kapitlen kan du skumma igenom någorlunda kvickt, för att få en ungefärlig uppfattning av innehållet.

H3. Kompendiet *Introduktion till LTH:s Linuxdatorer* får användas under laborationen, och hänvisas till i uppgifterna. I hänvisningarna förkortas namnet på kompendiet till ILL.

Länkar till det ovan refererade materialet finns på kurshemsidan under ”Datorlaborationer” om du inte har pappersversionerna av dem.

Kontrollfrågor

K1. Hur ser kommandot ut som loggar in på en annan dator via nätet (t.ex. om du vill logga in hemifrån)?
K2. Hur gör man för att byta lösenord på studentdatorerna?
K3. Vad betyder kommandot `pwd`?
K4. Vad är en kommandotolk?
K5. Hur får man tillbaka ett tidigare givet kommando, så man kan köra det igen?
K6. Hur kan man få hjälp om användningen av ett kommando, förutsatt att man vet namnet på kommandot?
K7. Hur skriver man ut en innehållsförteckning över den aktuella katalogen?
K8. Vilka är kommandona för att skapa respektive ta bort kataloger?
K9. Hur byter man aktuell katalog?
K10. Vad betyder tecknen `?` och `*` när man skriver dem på en kommandorad?
K11. Vad betyder tecknen `<` och `>` när man skriver dem på en kommandorad?
K12. Förklara kort hur systemet med åtkomsträttigheter av filer och kataloger fungerar.
K13. Vilket kommando utnyttjar man om man vill titta på innehållet i en fil en sida i taget?
K14. Antag att programmet `prog` producerar många sidor utskrift. Hur gör man för att titta på utskriften en sida i taget?
K15. Vad är en process?
K16. Hur avbryter man ett exekverande program?

Datorarbete

Kom ihåg att laborationen är ett inlärningsmoment. Ta hjälp av labbledaren om du har frågor, och anteckna gärna frågor som du vill diskutera med handledaren under din redovisning.

D1. Logga in på datorn. Använd det användarnamn och det lösenord som du tidigare har kvitterat ut (ILL 1.3).

D2. Fönsterhantering (ILL 1.4). Ägna några minuter åt att bekanta dig med fönstermiljön. Klicka runt bland menyer och applikationer.

D3. Editering av kommandoraden och enkla Unix-kommandon.

a) Öppna ett kommandofönster (Terminal).
b) Skriv några enkla Unix-kommandon, till exempel `pwd`, `ls`, `date` och `cal`.
c) Skriv avsiktligt fel och rätta felet. Prova specialtecknen för att radera enstaka tecken på kommandoraden och för att radera hela raden (ILL 3.3).
d) Skriv ut hjälptexten (”man-sidan”) för `date`-kommandot (ILL 1.8). Bläddra framåt och bakåt i texten.
e) Prova specialtecknen för att få tillbaka tidigare kommandon. Återkalla t.ex. kommandot för utskrift av datum och utför det på nytt. Prova både piltangenterna ↑ ↓ och CONTROL-R. Du kan också prova kommandot `history` för att se vilka kommandon du har kört.
f) Använd kalenderprogrammet (`cal`) för att ta reda på vilken veckodag du är född.
g) Intresserade kan prova `!` (s.k. *history expansion*) för att repetera tidigare kommandon. Exempelvis:

```
ls -l      // lista filer, med mer info
!!         // repetera det senaste kommandot
man touch  // visa manualen för touch
touch fil  // skapa en fil som heter 'fil'
!ls        // repetera det senaste kommandot som börjar med 'ls'
```

D4. Kommandon för att hantera filer och kataloger (ILL 1.7, 2.2, 2.6–2.9).

a) Skriv ut en innehållsförteckning över din hemkatalog. Skriv ut en förteckning där också filer vars namn börjar med punkt (s.k. *punktfiler*) skrivs ut.
b) Gå till katalogen `/usr/local/cs/dod/me/metool/src/metool`. Skriv ut en innehållsförteckning över katalogen. Skriv ut en förteckning över de filer vars namn innehåller strängen `Statement`.
c) Prova hur filnamnskomplettering fungerar. Skriv `less R` och tryck på TAB. Datorn fyller i tecken i filnamnet så länge de är unika (nu står det `less Re` på kommandoraden). Det finns mer än en fil vars namn börjar med `Re`. Tryck på TAB en gång till (ibland behövs det två extra tryckningar) så får du en lista över dessa filer. Skriv `a` och tryck på TAB igen; datorn fyller i till det unika filnamnet `ReadStatement.java`. Tryck på RETURN för att titta på filen.
d) Gå till din hemkatalog och skapa en katalog för de filer som används i denna laboration. Katalogen ska heta `lab1` och vara en underkatalog till en katalog `dod`, där du ska spara allt som rör kursen Datorer och datoranvändning. Du ska i fortsättningen skapa en ny katalog för varje datorlaboration som du gör. Använd följande kommandon:

```
cd          // gå till hemkatalogen om du inte redan är där
```

```
mkdir dod   // skapa katlogen dod i din hemkatalog
cd dod      // gå till katalogen dod
mkdir lab1  // skapa katalogen lab1
cd lab1     // gå till katalogen lab1
```

e) Kopiera filen `/usr/local/cs/dod/lab1/example.txt` till katalogen `lab1`. Skriv ut filen på skärmen med en sida i taget.

f) Undersök hur mycket utrymme du har tillgängligt för att lagra filer.

g) Tag reda på hur stor filen `example.txt` är. Komprimera därefter filen och tag reda på storleken hos den komprimerade filen. Återställ sedan filen till sitt ursprungliga utseende.

h) Skriv ut de rader i filen `example.txt` som innehåller ordet Unix. Kommandot för att leta i en fil heter `grep`.

i) Samma som uppgift h), men koppla om utskriften så att den hamnar i en fil med namnet `unix.txt`. Skriv ut denna fil på skärmen.

j) Räkna (med ett kommando) antalet rader i filen `unix.txt`. Du har nu räknat antalet rader som innehåller ordet Unix i filen `example.txt`.

k) Tag bort filen `unix.txt`.

l) Gör samma sak som i uppgift i)–k) utan att använda en temporär fil. Koppla i stället ihop kommandona med en pipe (|).

D5. Editering av text. På LTHs Linuxdatorer finns flera editorer, till exempel `nano` (enkel, terminalbaserad), `gedit` (enkel, fönsterbaserad), `code` (enkel, fönsterbaserad) och `emacs` (avancerad). Du får naturligtvis använda vilken editor du vill normalt, men här ska du testa `gedit`. Gör gärna om uppgifterna nedan i någon annan editor senare, på egen hand. Vissa uppgifter kommer framstå som väldigt enkla, men prova gärna att göra dem t.ex. i `nano`.

a) Starta `gedit` och läs in filen `example.txt` genom att i terminalfönstret skriva:
`gedit example.txt &`

b) Utnyttja musen och piltangenterna för att flytta textmarkören. Ändra textinnehållet genom att ta bort tecken och skriva in tecken. Spara det ändrade innehållet till filen.

c) Kontrollera att filen `example.txt` har ändrats.

d) Dela en rad i två rader. Sätt ihop raden igen.

e) Lägg in några tomma rader, tag sedan bort dem igen.

f) Utnyttja rullningslisten för att flytta dig i texten. Gå till början av texten. Gå till slutet av texten. Gå till rad 43 i texten.

g) Markera ett textblock genom att trycka på vänster musknapp och dra markören.

h) Markera ett textblock genom att först flytta markören till början av blocket (med musen eller tangentbordet), sedan håll ned Skift-tangenten, och flytta markören till slutet av textblocket (med musen eller tangentbordet).

i) Experimentera gärna också med tangenterna `Home` och `End`, och med `Ctrl` och piltangenterna ← → för att flytta markören och markera text.

j) Kopiera ett markerat textblock till en annan plats i filen. Flytta sedan ett markerat textblock.

k) Gå till början av filen och leta upp den första förekomsten av ordet Unix. Leta sedan upp nästa förekomst, osv. Byt sedan alla Unix mot Xinu.

D6. Hantering av processer (ILL 3.3, 3.8).

a) Skriv kommandot `xeyes` i terminalfönstret. Flytta musen så ser du att ögonen följer musmarkören. Notera att man inte kan fortsätta att skriva kommandon i fönstret eftersom det är låst av `xeyes`-programmet.

b) Skriv CONTROL-C i kommandofönstret för att avbryta `xeyes`-programmet. Programmets fönster försvinner när man avbryter programmet.

c) Skriv nu `xeyes &` i terminalfönstret. `&`-tecknet betyder att programmet ska köras som en självständig process som inte är kopplad till terminalfönstret. Nu kan man alltså fortsätta att skriva kommandon i terminalfönstret. Avsluta `xeyes` genom att högerklicka med musen på ikonen för `xeyes` i verktygsraden på skärmens vänstra sida. Välj `Quit` i menyn som visas.

d) Man kan tillfälligt avbryta exekveringen av ett program med CONTROL-Z. Skriv `xeyes` och sedan CONTROL-Z. Notera att programmet nu inte är aktivt (ögonen följer inte musmarkören). Med kommandot `fg` (foreground) återupptar man exekveringen igen. Om man i stället ger kommandot `bg` (background) återupptar man exekveringen "i bakgrunden", precis som om man hade startat programmet med `xeyes &`.

D7. Inloggning på andra datorer (ILL 3.7).

a) Prova att logga in på datorn `login.student.lth.se` med hjälp av kommandot `ssh`. Prova att ge några kommandon, t.ex. `touch` för att skapa en ny fil. Avsluta genom att skriva `exit`.

b) Datorn `login.student.lth.se` är på samma nätverk som datorerna i datorsalarna. Med `ssh` loggade du in på en annan dator, men fortfarande med ditt egna konto. Om du skapade en ny fil så kommer du fortfarande hitta den i din lokala terminal efter att du har avslutat `ssh`.

c) Datorn `login.student.lth.se` kan nås externt, t.ex. om du behöver logga in hemifrån.

D8. Glöm inte att logga ut innan du lämnar datorn!

Laboration 2 — LaTeX

Mål: Du ska lära dig grunderna i LaTeX och tillämpa dina kunskaper på ett exempel.

Hemarbete

H1. Titta igenom föreläsningsbilderna till föreläsningen om LaTeX.

H2. Läs igenom kompendiet *Att skriva rapporter med LaTeX*, åtminstone så mycket så att du blir bekant med vad man kan göra med LaTeX. Du behöver inte försöka memorera alla detaljer.

H3. Med början på nästa sida finns ett exempel på en rapport som är producerad med LaTeX. Studera rapporten och försök komma på vilka kommandon som behövs för att få texten att se ut som den gör. Markera i rapporten, eller i ett textdokument, vilka kommandon du tänker använda.

Kontrollfrågor

K1. Laborationsledaren kontrollerar att du gått igenom exempelrapporten och markerat vilka LaTeX-kommandon du ska utnyttja för att formatera texten.

Datorarbete

Notis: I instruktionerna nedan föreslår vi att ni använder programmet Texmaker för att arbeta med LaTeX, men ni får lov att använda vilken editor ni vill. Det går också bra att använda onlineverktyg, så som Overleaf.

D1. Skapa en ny katalog med namnet `dod/lab2` och gå till denna katalog.

D2. Kopiera filen `/usr/local/cs/dod/lab2/rapportmall.tex` till din katalog. Ge den kopierade filen namnet `rapport.tex`. Filen innehåller en LaTeX-mall för rapporter, liknande den mall som beskrivs i avsnitt 2.2 i LaTeX-kompendiet och i föreläsningsbilderna.

D3. I filen `/usr/local/cs/dod/lab2/rapporttext.txt` finns texten till rapporten som beskrivs i uppgift H3, utan några LaTeX-kommandon och utan några figurer. Lägg in innehållet i denna fil i din `.tex`-fil, mellan `\begin{document}` och `\end{document}`.

D4. Starta Texmaker med kommandot `texmaker &` (eller `texmaker rapport.tex &`).

D5. Lägg in lämpliga LaTeX-kommandon i filen så att rapporten får (åtminstone ungefär) det utseende som beskrivs i uppgift H3. Se till att styckeindelningen och rubrikerna blir korrekta i hela dokumentet innan du ger dig på resten, till exempel de matematiska formlerna.

Arbeta stegvis: ändra lite, klicka på pilen till vänster om Quick Build så körs pdfLaTeX, titta på resultatet, ändra lite till, osv.

Bilderna som ska inkluderas i dokumentet finns i filerna `nrbild.pdf` och `konvbild.pdf` i katalogen `/usr/local/cs/dod/lab2/`, programkoden finns i filen `NewtonRaphson.java` i samma katalog.

D6. Prova gärna att skriva ut rapporten på skrivaren när du är nöjd med dokumentets utseende (ej obligatoriskt).

D7. Om du har tid: prova sådana möjligheter i LaTeX som du inte har behövt använda tidigare: listor av olika slag, innehållsförteckning, mera avancerade formler, osv.

Newton-Raphsons metod

Per Holm

21 maj 2001

1 Numeriska metoder

En *numerisk metod* är en metod med vars hjälp man kan finna en approximativ lösning till ett matematiskt problem. Lösningen ges i numerisk form dvs i form av ett antal talvärden. Till exempel kan man med en numerisk metod räkna ut att roten till ekvationen $x^2 = 3$ är $1.73205\ldots$, men inte få fram den analytiska lösningen $\sqrt{3}$.

I många problem är man hänvisad till numeriska metoder. Det kan bero på att problemet inte har någon analytisk lösning eller att det skulle vara alltför arbetsamt att få fram denna.

I denna rapport studeras en numerisk metod för att lösa ekvationer med en obekant, nämligen Newton-Raphsons metod.[1]

2 Newton-Raphsons metod

2.1 Teori

Vi söker en rot α till ekvationen $f(x) = 0$. Vi antar att vi vet att roten ligger i närheten av talet x_0 och ska försöka förbättra denna approximativa lösning. Vi kan bilda nya approximationer $x_1, x_2, x_3, \ldots$ med följande formel:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{1}$$

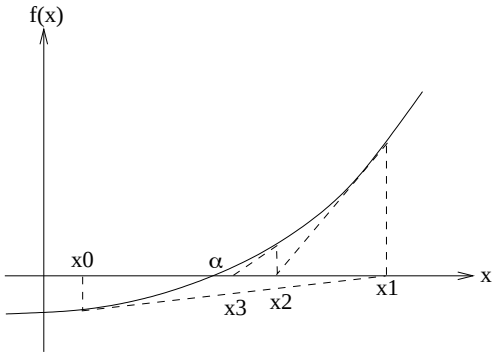
Metoden åskådliggörs geometriskt i figur 1. Tangenten i punkten $(x_0, f(x_0))$ skär x-axeln för $x = x_1$, tangenten i punkten $(x_1, f(x_1))$ skär x-axeln för $x = x_2$, osv. Av figuren förstår vi att $x_k \to \alpha$ då $k \to \infty$.

Newton-Raphsons metod kan härledas genom att man serieutvecklar funktionen $f(x_0 - \epsilon)$ och trunkerar utvecklingen efter den linjära termen. Ur serieutvecklingen kan man också få ett uttryck på feltermen.

2.2 Exempel

Ekvationen $e^{-x} = \sin x$ har en rot nära 0.6. Bestäm denna rot med Newton-Raphsons metod.

[1] Metoden kallas ibland Newtons metod.



Figur 1: Newton-Raphsons metod

Här är $f(x) = e^{-x} - \sin x$, $f'(x) = -e^{-x} - \cos x$. Om vi räknar med 9 decimaler så får vi:

x	$f(x)$	$f'(x)$	$f(x)/f'(x)$
0.6	-0.015830837	-1.374147251	0.011510481
0.588479519	0.000073820	-1.386956425	-0.000053224
0.588532743	0.000000001	-1.386897331	-0.000000001
0.588532744			

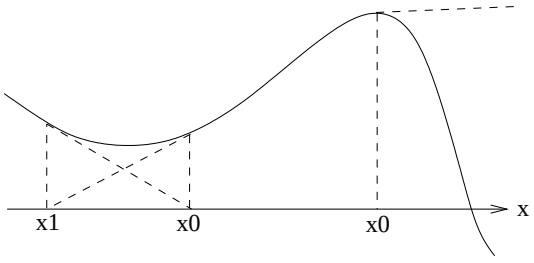
dvs $\alpha = 0.588532744$.

2.3 Konvergens

Newton-Raphsons metod konvergerar i allmänhet *kvadratiskt* mot den sökta roten, vilket innebär att antalet korrekta siffror i svaret fördubblas i varje iteration. Men det finns fall då konvergensen blir sämre eller då metoden inte alls konvergerar, nämligen då man råkar ut för en derivata som är nära noll. Två sådana fall illustreras i figur 2.

3 Newton-Raphsons metod i Java

Det är inte helt enkelt att implementera Newton-Raphsons metod i ett program, åtminstone inte om man kräver att metoden ska fungera i alla upptänkliga fall. I



Figur 2: Konvergensproblem

metoden `solve` som visas nedan finns bara den grundläggande algoritmen med: vi har inte tagit hänsyn till undantagsfall som finns eller fel som kan inträffa, till exempel att derivatan `fprim(x0)` blir nära noll (se avsnitt 2.3).

```
class NewtonRaphson {
    private static double f(double x) {
        return ...;
    }

    private static double fprim(double x) {
        return ...;
    }

    public static double solve(double x0, double eps) {
        double x1 = x0;
        do {
            x0 = x1;
            x1 = x0 - f(x0) / fprim(x0);
        } while (Math.abs(x1 - x0) > Math.abs(x1) * eps);
        return x1;
    }
}
```

Laboration 3 — Versionshantering med Git och GitHub

Mål: Du ska bekanta dig med grundläggande tekniker för versionshantering och att arbeta flera personer i samma utvecklingsprojekt med hjälp av Git och GitHub. Laborationen kommer att utgöra en guidad tur genom ett enkelt användningsfall. Syftet är alltså inte att bli experter, utan att få se och senare kunna känna igen kommandon och få en känsla för arbetssättet.

Hemarbete

H1. Titta igenom föreläsningsbilderna från föreläsning 2 (versionshantering med Git och GitHub).

H2. Läs igenom *Appendix G Versionshantering och kodlagring* i *Introduktion till programmering med Scala* av Björn Regnell.

H3. Se avsnitt 1.1, 1.2, 1.3, 1.6 och 1.7 ("fäll ner" flikarna uppe till höger för att komma åt de individuella avsnitten) i youtube-serien *Git and GitHub for Poets*.

H4. Läs igenom kapitel 1, 2 och 3.1-3.2 i *Pro Git* av Scott Chacon och Ben Straub. Vi kommer inte att använda oss av s.k. *branches* (grenar) direkt i laborationen, men det kommer att bli ett viktigt begrepp i era senare kurser och introducerar begreppet *merge* (sammanfogning) som ni stöter på i laborationen.

H5. Gå till GitHub (`https://github.com`) och skapa dig ett eget gratiskonto. Försök välja ett kontonamn du känner att du kommer att vara bekväm med för en lång tid framöver. Det är mycket möjligt du kommer att ha kvar det under hela din yrkesverksamma tid. Prova till exempel `fornamnefternamn` utan svenska tecken och se om det är ledigt.

H6. Läs igenom uppgifterna under rubriken Datorarbete.

Länkar till det ovan refererade materialet finns på kurshemsidan under "Datorlaborationer" om du inte har pappersversionerna av dem.

Kontrollfrågor

K1. Vad är ett versionshanteringssystem?
K2. Vad är skillnaden mellan Git och GitHub?
K3. Vad är ett repositorium (eng. *repository*).
K4. Vad innebär det att göra en klon av ett repositorium?
K5. Vad innebär begreppet *commit*?
K6. Vad är skillnaden mellan commit och push i Git?
K7. Vad är skillnaden mellan push och pull i Git?
K8. Vad är en sammanfogningskonflikt (eng. *merge conflict*)?
K9. Ge exempel på en situation då en sammanfogningskonflikt (eng. *merge conflict*) kan uppstå.

Datorarbete

Under laborationen kommer vi att gå igenom ett scenario steg för steg där vi: 1) Skapar ett projekt, även kallat repositorium (*repository* på engelska) eller i dagligt tal förkortat till *repo*, på GitHub; 2) Skapar en lokal klon av repositoriet på vår lokala dator och arbetar med det; 3) Kopierar upp resultatet av vårt arbete till det centrala repositoriet på GitHub. Vi kommer därefter simulera fallet att två utvecklare arbetar tillsammans med samma projekt och har vars sin lokala kopia av repositoriet på sin egen dator.

Observera att några uppgifter beskriver vad du ska göra men inte exakt hur. I slutet av handledningen finns lösningsförslag men försök att själv leta reda på en lösning med hjälp av t.ex. kursmaterialet, ett ”cheat sheet” eller genom att söka på Internet.

D1. Logga in på ditt konto på GitHub och skapa ett nytt repositorium för laborationen. Klicka på den gröna knappen med texten ”New” uppe till vänster på sidan du kommer till när du loggat in.

Ge repositoriet namnet ”edaa60-lab3” och ange att det ska vara privat (*private*). Klicka även i rutan framför ”Initialize this repository with a README”. Det senare innebär att det redan från början finns ett innehåll i repositoriet vilket underlättar när vi sedan ska skapa vår lokala klon på vår egen dator. I drop-down-menyn ”Add .gitignore:” väljer du ”Scala”. Vi ska nämligen utveckla ett enkelt system i Scala och filen .gitignore talar bland annat om för Git att inte bry sig om de filer som scalakompilatorn genererar. Den är inte nödvändig, men gör att vi slipper lite distraherande meddelanden från Git senare. Klicka på ”Create repository” för att skapa repositoriet.

Du kan du nu redigera filen README.md genom att klicka på pennsymbolen efter filnamnet (eller genom att öppna filen genom att klicka på filnamnet i listan och därefter klicka på pennsymbolen till höger i fönstret). I README-filen brukar man skriva en beskrivning av vad repositoriet innehåller för något och saker man behöver veta för att komma igång med innehållet. Skriv något kort relaterat till laborationen.

När du är färdig sparar du filen genom att göra en så kallad *commit*. Varje commit bör medföljas ev en beskrivning av varför filen ändrades. Ofta räcker det med en kort kommentar. Skriv en sådan kort kommentar i rutan under rubriken ”Commit changes”. Du kan t.ex. skriva ”Provided initial description of the project”. Om en längre beskrivning är motiverad kan denna lämnas i rutan undertill. Klicka sedan på ”Commit changes”. Gå tillbaka till repositoriets huvudsida (klicka på den blå länken ”edaa60-lab3”) och betrakta resultatet.

Nu ska vi se hur vi kan arbeta med git på vår lokala dator!

D2. Innan vi börjar använda Git för första gången bör vi göra några mindre inställningar. Dessa behöver vi bara göra en gång. Det handlar om att tala om för Git vem vi är och vilken texteditor vi föredrar. Som standard använder git editorn `vim`, vilket de flesta nog inte vill, då den är känd för att vara väldigt kraftful men svåranvänd och ha en hög inlärningskurva.

`[C1]` Använd kommandot `git config` med flaggan `global` för att sätta ditt användarnamn, e-postadress och din föredragna editor. Du kan t.ex. använda editorn `nano` (kör direkt i terminalen) eller `gedit` (öppnas i eget fönster).

Editorn du anger kommer användas när du gör `git commit` i terminalen, där git ber dig att skriva en commit-kommentar. Att välja en editor som fungerar direkt i terminalen, snarare än att behöva öppna ett nytt fönster, kan då vara smidigt.

D3. Nu börjar vi närma oss att kunna göra en lokal klon av vårt repo och börja arbeta med git, men vi behöver sätta upp en sak till: en SSH-nyckel. I en alltmer osäker cyberrymd krävs

det numera att man kopplar upp sig till GitHub via protokollet SSH (som vi använde för att koppla upp oss mot annan skoldator i slutet av labb 1). För att slippa behöva uppge inloggningsuppgifter vid varje git-kommando används en nyckel (key eller key-pair) som måste genereras på den egna maskinen innan man kan klona repositoriet lokalt. Om du redan har satt upp en SSH-nyckel kan du hoppa över denna del.

De följande stegen är baserade på GitHubs egen guide "Generating a new SSH key"[1]:

1. Öppna terminalen
2. Skriv följande i terminalen:
   `ssh-keygen -t ed25519 -C "din_email@exempel.se"`
3. SSH ber dig att döpa filen, default namnet går bra att använda, tryck Enter.
4. Ingen passphrase behövs i detta läge, tryck Enter två gånger

Nu har du genererat ett så kallat nyckel-par med en privat och en publik nyckel (mer om detta i senare kurser, men för intresserade, se fotnot[2]). Gör `ls` på din dolda ssh-katalog för att se filerna:

```
> ls ~/.ssh
id_ed25519         # din privata nyckel
id_ed25519.pub     # din publika nyckel
known_hosts
```

Nu saknas endast att lägga till din publika nyckel i ditt GitHub-konto så att hemsidan kan autentisera dig. Följande guide beskriver de sista stegen under rubriken "Adding a new SSH key to your account": `https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account#adding-a-new-ssh-key-to-your-account`.

När alla stegen är gjorda på hemsidan ska det vara fritt fram att börja använda git och arbeta mot GitHub utan att behöva uppge användarnamn och lösenord. Var inte rädd att fråga din handledare om du fastnar vid något av stegen.

D4. Det är nu dags att skapa en lokal kopia, eller *klon*, av vårt repositorium på vår lokala dator. Skapa en katalog på lämpligt ställe med hjälp av kommandon i terminalfönstret, t.ex. `~/dod/lab3`. Gå sedan till katalogen du nyss skapade med hjälp av kommandot `cd`.

För att kunna klona vårt repositorium behöver vi en referens till det. Gå därför till webbsidan för ditt repositorium. Där hittar du en grön knapp med texten "Code". Om du klickar på denna får du upp en drop-downmeny som bland annat innehåller en URL, som du ska kopiera. Kontrollera att du tittar på SSH-varianten snarare än HTTS, URL:en ska vara på formatet `git@github.com:<username>/dod-lab3.git`.

`[C2]` Använd nu kommandot `git clone` för att klona repositoriet.

Du ska nu ha fått en katalog med samma namn som ditt repositorium. Gå ner i denna katalog och undersök innehållet med `ls -la`. Ser du README-filen och `.gitignore`? Dessutom borde du ha en katalog kallad `.git`. Det är här Git sparar allt den behöver veta för att hålla reda på våra filer. Du kan alltid kontrollera det aktuella tillståndet för dina projektfiler genom att skriva:

```
git status
```

[1] `https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent#generating-a-new-ssh-key`

[2] `https://en.wikipedia.org/wiki/Public-key_cryptography`

Kommandot borde bekräfta att alla dina filer stämmer överens med originalet på GitHub.

D5. Vi ska nu börja utvecklingen av vårt scalaprogram. Skapa en fil i katalogen, kalla den t.ex. `HelloWorld.scala`, med följande innehåll:

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, world!")
    }
}
```

Prova att kompilera och testköra programmet för att verifiera att det fungerar:

```
scalac HelloWorld.scala
scala HelloWorld
```

Kontrollera innehållet i katalogen och statusen för projektet:

```
ls
git status
```

Git har upptäckt att det finns en fil (`HelloWorld.scala`) som Git tror du kanske vill ha med i ditt projekt och föreslår att du ska lägga till den. Observera att Git ignorerar filerna som slutar på `.class`. Detta beror på att innehållet i `.gitignore` instruerar Git att ignorera dessa filer. Vilka andra filer ignoreras? Öppna `.gitignore` och titta efter!

För att Git ska hålla reda på vår nya fil måste vi instruera Git att addera den till sin databas. Det görs i två steg: Först väljer vi vilka filer det gäller genom att lägga till dem i s.k. *staging area* (`git add`). Därefter ger vi kommandot för att faktiskt lägga till (`git commit`). I vårt fall har vi bara en fil, så det första steget kan verka onödigt, men ibland vill man ha mer kontroll över vilka filer som ska eller inte ska läggas till, innan man gör en commit.

`[C3]` Lägg till din fil i staging area med kommandot `git add`, och kolla status igen efteråt.

Vi ser att Git har förberett för att spara filen i sin databas, men ännu inte gjort det. Det görs först när man gör en *commit*. Då sparas en permanent fryst version av den eller de aktuella filer som finns i staging area, tillsammans med användarens kommentar som beskriver syftet med ändringen som gjorts. Om det räcker med en kort kommentar (vilket det oftast gör) kan man skriva kommentaren direkt på kommandoraden med optionen `-m`. Om man inte använder optionen `-m` kommer den editor du angav när du tidigare konfiguerade Git att startas. En temporär fil som Git bevakar öppnas i editorn, som du kan skriva vad du vill i på vanligt vis, och när du är klar så sparar och stänger du filen. Detta är användbart när längre kommentarer krävs.

`[C4]` Spara den nuvarande versionen av din fil i databasen med kommandot `git commit`.

Vid utskrift av projektets status bör det nu framgå att alla filer i katalogen är synkroniserade med de som finns i (det lokala) repositoriets databas.

Du kan nu fortsätta att arbeta med scalaprogrammet i `HelloWorld.scala`. Ändra programmet med din favoriteditor så att något annat än ”Hello, world!” skrivs ut, spara, kompilera och testkör.

När du är nöjd med din ändring undersöker vi projektets status igen:

```
git status
```

Vi ser att Git har upptäckt att `HelloWorld.scala` har modifierats. För att lagra den nya versionen i Git-databasen gör vi likadant som när vi adderade filen, d.v.s. först `git add` och sedan `git commit`. Man kan också skriva `git add .` (notera punkten) för att addera alla ändrade filer. Många gånger vill man addera och committa alla modifierade filer. Därför finns det en genväg för att slippa göra både *add* och *commit*. Man kan använda optionen `-a` på commit-kommandot (`git commit -a`) för att automatiskt göra *add* på alla modifierade filer. Notera att detta bara är sant för filer som redan har blivit tillagda i Git. För nyskapta filer behöver man således manuellt köra *add* första gången.

`[C5]` Välj en av metoderna och gör commit på dina ändringar! Kontrollera med `git status` att ditt lokala repositorium är uppdaterat.

D6. De ändringar vi nu gjort finns bara lagrade i vårt lokala repositorium på vår egen dator. För att kunna dela med oss av vårt uppdaterade projekt till andra, eller för att vi själva ska kunna komma åt projektet från andra platser, vill vi ladda upp våra ändringar till den centrala lagringsplatsen på GitHub.

Prova det nu:

```
git push
```

Efter att du kört kommandot så gå tillbaka till ditt projekt på GitHub och ladda om den. Undersök vilka förändringar som skett i projektet. Klicka på `HelloWorld.scala` och kontrollera innehållet. Klicka på ”History” och undersök vilka versioner av filen som finns sparade samt vad de innehåller.

D7. En av de stora vinsterna med att använda Git är att det underlättar mycket när flera personer arbetar tillsammans på samma projekt, eller om en och samma person vill arbeta med olika saker på projektet parallellt – på en och samma eller flera datorer. Om man samarbetar med en annan person behöver man ge den personen rättighet att arbeta med projektet på GitHub. För att bjuda in en annan person till projektet så klicka på ”Settings” på projektets webbsida och välj därefter ”Manage access” i menyn till vänster. Här har du möjlighet att bjuda in en annan GitHub-användare till projektet. Klicka dig dit för att se hur sidan ser ut.

I fortsättningen av laborationen kommer vi att simulera en sådan parallell utveckling genom att du själv skapar en andra klon av repositoriet och arbetar med den parallellt med den klon vi tidigare skapade. Vi kommer också att se exempel på hur man kan arbeta med Git inifrån ett utvecklingsverktyg utan att använda sig av terminalen.

Som utvecklingsverktyg kommer vi använder oss av editorn Visual Studio Code (i fortsättningen förkortat *VS Code*). VS Code är en generell texteditor med stöd för många olika programmeringsspråk och som även har inbyggt stöd för Git. I VS Code ska vi klona ditt repositorium igen, och alltså ha två kloner på olika ställen på din dator; en som vi arbetar med i terminalen och en som vi arbetar med i VS Code.

Starta editorn i terminalen genom att skriva:

```
code
```

Efter en kort stund ska ett nytt fönster öppnas. För att skapa en ny klon av ditt repositorium på GitHub kan du trycka CTRL+SHIFT+P. Då får du upp något som kallas för ”kommandopaletten”. Sök eller scrolla ner i listan tills du hittar kommandot ”Git: Clone” och välj detta. Skriv sedan in URL:en till ditt repositorium på GitHub (samma som när

du gjorde `git clone` i terminalen tidigare) och tryck ENTER. Du får nu upp en fildialog för att välja var du vill placera det nya klonade repositoriet. Skapa en ny katalog bredvid katalogen du skapade tidigare (`lab3`) kallad t.ex. `lab3b` och välj denna som målkatalog. Om allt gått väl kommer VS Code att skapa en ny klon och du får en dialogruta som frågar om du vill öppna det nya repositoriet. Välj ”Open”!

D8. Längst ut åt vänster finns en rad symboler som representerar olika verktyg. Den översta kallas för ”Explorer” och används för att se vilka filer som finns i katalogen och för att editera dessa. Gå in i filen `HelloWorld.scala` och ändra återigen utskriften så att något nytt skrivs ut samt spara filen.

Den tredje symbolen representerar verktyget `Source Control`. Det är ”Git-fliken”, klicka på denna! Du ser då att filen `HelloWorld.scala` är markerad som modifierad (”changed”) och till höger om denna finns en rad små ikoner. Med hjälp av dessa kan du i tur och ordning öppna filen, ångra ändringarna och göra något som kallas för att ”stage:a” filen, alltså lägga till den i staging area, precis som vi gjorde tidigre genom att köra `git add`. Klicka på ikonen i form av ett litet plustecken för att göra ”add” på filen. Dess tillstånd ändras nu till ”staged”.

Vi är nu nöjda med våra ändringar och det är dags att göra commit på dem. En bit upp i fönstret, strax till höger om rubriken ”SOURCE CONTROL” hittar du ytterligare tre ikoner. Tryck på den första (en bock) för att göra commit. Fyll i en commit-komentar i dialogrutan som dyker upp.

D9. Till vänster om commit-ikonen hittar du en ikon som består av tre punkter. Klickar du på den dyker en pull-down-meny upp med ytterligare Git-kommandon. Välj nu ”Push” i menyn för att ladda upp din ändring till GitHub. Kontrollera på projektets webbsida att ändringen är synlig.

D10. Nu är dina ändringar överförda till det centrala repositoriet på GitHub, men har ännu inte överförts till din första klon av repositoriet – det som du tidigare har jobbat mot via terminalfönstret. För att göra detta används kommandot `git pull`.

`[C6]` Gå till terminalfönstret och hämta ner de senaste ändringarna från det centrala repositoriet med git pull.

Kontrollera att ändringarna du gjorde har laddats ner genom att inspektera innehållet i filen `HelloWorld.scala`.

D11. Så här långt har vi, trots att vi har separata kloner av vårt repositorium, hela tiden arbetat *sekvensiellt* och vid varje förändring sparat undan våra ändringar till det centrala repositoriet innan vi har gjort någon förändring i den andra klonen. Men den stora poängen med Git och versionshantering är att det stöder *parallellt* arbete i de olika klonerna. Vilka problem stöter vi på då? Låt oss prova och se vad som händer.

`[C7]` I ditt först klonade repositorium (det som du arbetar mot via terminalen):

a) Ändra återigen `HelloWorld.scala` så att en ny text skrivs ut (ändra på raden med `println`-kommandot).

b) Gör ”stage” (`git add`) samt commit (`git commit`), eller använd flaggan `-a` till commit-kommandot för att göra båda stegen samtidigt.

c) Ladda upp din ändring till GitHub (`git push`)

Nu ska vi prova att göra en motsvarande ändring i det andra repositoriet:

a) Gå till ditt VS Code-fönster.

b) Gör en ändring på samma rad i `HelloWorld.scala` som du ändrade nyss så att en ny text skrivs ut.

c) Spara filen.

d) Gör ”stage” på filen (den lilla plusikonen efter filnamnet i ”changed”-listan).

e) Gör commit på dina ändringar (den lilla checkikonen längst upp).

f) Prova att ladda upp dina ändringar till GitHub (”Push” i pull-down-menyn du får när du klickar på de tre prickarna till höger om ”commit”-ikonen). Vad händer?

Uppenbarligen har någon (faktiskt vi själva i detta fallet) gjort ändringar parallellt med våra, och push:at dem till GitHub före oss. Git tillåter oss då inte att göra push, för då hade våra ändringar skrivit över de ändringarna som andra personer har gjort. Istället måste vi först ladda ner och foga samman de ändringar som andra gjort med våra egna (och kontrollera att de fungerar ihop) innan vi kan ladda upp dem till GitHub:

a) Klicka bort dialogrutan med felmeddelandet.

b) Välj ”Pull” i pull-down-menyn (de tre prickarna...).

De senaste ändringarna från GitHub-repositoriet laddas nu ner och sammanfogas (eng. *merge*) med de vi gjort lokalt. I många fall klarar Git av att göra detta helt på egen hand och det enda vi behöver göra är att kontrollera att vårt program fortfarande går att kompilera och beter sig som vi förväntar oss. Sedan kan vi på nytt ladda upp ändringarna till GitHub. Ibland klarar dock inte Git av att göra sammanfogningen på ett entydigt sätt utan vi måste ingripa och göra sammanfogningen manuellt. Det har uppstått en så kallad *sammanfogningskonflikt* (eng. *merge conflict*). En sådan situation uppstår när ändringar gjorts på samma rader i respektive version av filerna. När en sådan här konflikt uppstår markerar Git konflikten genom att skjuta in extra rader i filen som beskriver konflikten och visar hur de olika versionerna av filen skiljer sig åt. Det kan t.ex. se ut så här:

```
<<<<<<< HEAD (Current Change)
println("Hello from version one!")
=======
println("Hello from the second version!")
>>>>>>> 7e01181febacf80fdd666e36d4d492aea0b0f620 (Incoming Change)
```

Det är nu vår uppgift att editera filen så att den blir så som vi vill ha den och ta bort de extra raderna som Git har skjutit in för att markera konflikten. Det kan vi göra genom att välja att behålla endera av versionerna, behålla båda, eller skriva om programmet på önskat vis. Om du gör detta i VS Code underlättar editorn arbetet genom att visa en rad med alternativ omedelbart ovanför konflikten med klickbara alternativ för de vanligaste valen, t.ex. för att behålla endera av versionerna. Lös konflikten på lämpligt sätt och ladda upp ändringarna till GitHub:

a) Välj en av versionerna genom att klicka på snabblänkarna eller editera filen manuellt.

b) Spara filen!

c) Gör ”stage” (add) på filen.

d) Gör commit.

e) Ladda upp ändringarna till GitHub genom att göra push[3]

f) Kontrollera förändringen i ditt projekts webbsida på GitHub.

[3] Om man inte har gjort exakt enligt instruktionen, tex glömt att spara filen innan man gör den staged, kan man råka ut för något som verkar vara en bugg i VS Code när man gör push som gör att push misslyckas därför att VS Code påstår att det finns konflikter kvar att lösa trots att dessa redan är lösta. Om detta händer så gör en valfri ändring i filen, t.ex. lägg till en kommentar, spara filen, gör filen staged (ej genom att göra commit på allt eller att göra alla filer staged i en klump), gör commit och prova att göra push igen. Du kan alltid ta bort den extra ändringen senare.

D12. Om du har tid: Gå till terminalfönstret och upprepa föregående deluppgift och prova på att på nytt lösa en sammanfogningskonflikt, fast denna gång genom att använda terminalkommandon. Dvs, ändra utskriften genom att ändra i `HelloWorld.scala`, gör add/-commit/push, studera felmeddelandet, ladda ner ändringar med pull, lös konflikten och gör add/commit/push på nytt.

Du har nu lärt dig tillräckligt mycket om Git för att kunna använda det för dina egna mjukvaruprojekt eller projekt du gör tillsammans med en eller några få kamrater. Under utbildningens gång kommer du att stöta på mer avancerade sätt att använda Git och hur det används för att hantera större mjukvaruprojekt.

Git-kommandon

Här ges lösningar till de uppgifter som är markerade. Försök gärna lösa uppgifterna utan denna hjälp först. Ta gärna hjälp av en "cheat sheet".

C1 Byt ut `fornamnefternamn` och `mejladr@plats.se` nedan mot ditt GitHub användarnamn eller ditt namn ("Förnamn Efternamn"). Om du föredrar att använda en annan texteditor än `nano` byter du även ut `nano` mot namnet på den editor du vill använda, t.ex. `gedit`

```
git config --global user.name fornamnefternamn
git config --global user.email mejladr@plats.se
git config --global user.editor nano
```

C2 Byt ut URL:en mot din egen:

```
git clone https://github.com/fornamnefternamn/edaa60-lab3.git
```

C3 Lägg till fil till *staging area*:

```
git add HelloWorld.scala
git status
```

C4 Spara permanent det som ligger i staging area:

```
git commit -m "Created initial version of the program"
```

C5 Gör t.ex.:

```
git commit -a -m "Changed program to make it more awesome"
```

C6 Man kan ge argument för att bestämma var ifrån man vill hämta ändringar, men i de flesta fall, liksom nu, räcker det att du kör:

```
git pull
```

C7 Gör t.ex.:

a) `nano HelloWorld.scala`

b) `git commit -a -m "Improved the print!"`

c) `git push`

Datorer och datoranvändning, godkända laborationsuppgifter

Efter du blivit godkänd på en laboration kan du be din handledare att signera nedan. Detta är inget som krävs – handledaren kommer anteckna ditt godkännande och föra in i vårt betygsystem – utan snarare en liten försäkring för din egen skull. Om det skulle ske ett misstag någonstans och ditt godkännande inte förs in i betygsystemet, kan du visa att du blivit godkänd.

Skriv ditt namn och din namnteckning nedan:

Namn: .....

Namnteckning: .....

Godkänd laboration	Datum	Laborationsledarens namnteckning
1 Linux		
2 Latex		
3 Git		