KTH Computer Science
and Communication

Gradvis typat Python med Cython

Examensarbete för kandidatexamen i datateknik - DD143X

FREDRIK GUSTAFSSON <FRGUSTAF@KTH.SE>
BÅTSMANSKLEVET 9, 12940 HÄGERSTEN
TEL.NR: 0702370163

VIKTOR HOLMBERG <VHO@KTH.SE>
BRANTBACKEN 3, 17832 EKERÖ
TEL.NR: 0704220660

Handledare: Mads Dam
Examinator: Mads Dam

Stockholm, Sverige 2011-03

Referat

I denna rapport undersöks Cython, ett experimentellt programmeringsspråk vilket tillför viss funktionalitet till Python, bland annat valfria statiska typannotationer. Huvudfrågan i denna rapport är vilken prestandapåverkan denna statiska typinformation har, men även dess effekter på typsäkerhet och mjukvarukonstruktionsaspekter i Python berörs i korthet.

För att utröna på vilket sätt typannotationerna påverkar prestanda utformades tre testfall. Testfallen typades till ett antal versioner med olika grad av statisk typning, varpå exekveringstiden mättes.

Gradvis typning med Cython visade sig vara ett utmärkt sätt att optimera vissa Pythonkod, särskilt numeriska beräkningar. Cythons valfritt statiska typsystem är dock inte ett fullgott alternativ till helt statiskt typade språk ur ett mjukvarukonstruktionsperspektiv.

Abstract

Gradually typed Python using Cython

In this report Cython, an experimental programming language which adds functionality to Python was studied. One of the functionalities added by Cython is optional static type annotations. How these optional static type declarations affect performance is the main focus of this report. The effect of these type annotations on type safety and software design aspects of Python was also studied, but to a lesser extent.

To establish how static type annotations affect performance three test cases were designed. These test cases were then gradually typed in a number of versions with different amounts of static typing. Finally, the execution times of the different versions were measured.

It was found that gradual typing using Cython is a excellent optimization strategy for some Python code, particularly numeric calculations. However, Cythons gradually typed type system can not fully be considered an alternative to popular staticly typed languages with respect to software design considerations.

Förord

Detta är en kandidatexamensrapport för kursen DD143X genomförd vårterminen 2011 på KTH, Stockholm. Rapporten och det bakomliggande arbetet genomfördes i sammarbete mellan båda författarna som bidrog med lika stora delar.

Tack till vår handledare Mads Dam för vägledning och rådgivning under projektets gång.

Innehåll

Bilagor

A Kod

B Mätdata

C Diagram

Kapitel 1

Inledning

Python är idag ett ett populärt programmeringsspråk. Dess filosofi att lämna många uppgifter till exekveringsmiljön vilka traditionellt utförs av programmeraren samt ett fokus på läsbarhet och koncis kod resulterar i ett språk mycket väl lämpat för skriptning och snabb utveckling av program.

Enkelheten i användningen av Python kommer dock till ett pris; i många fall har Python en långsam exekveringshastighet jämfört med mer maskinnära språk som C, C++ eller Java. Detta är en följd av att Python hanterar många funktioner i körtid vilka normalt hanteras vid kompilering, till exempel typkontroll eller metoduppslagning. Ett annat problem är att Pythons dynamiska typsystem ökar risken för att typfel inte upptäcks direkt, utan ligger kvar i koden och uppstår vid ett senare tillfälle då de kan vara svårare att åtgärda.

I denna rapport studeras Cython, ett experimentellt tillägg till Python som har potentialen att lösa dessa problem. Särskilt undersöks Cythons gradvis typade typsystem, vilket tillåter programmeraren att vid prestanda- eller säkerhetskritiska delar av ett program införa valfri statiska typannotationer. Dessa kan sedan användas av Cython för att optimera prestanda och kontrollera typsäkerhet vid kompilering. Möjligheten att på detta sätt endast optimera de delar av programmet som verkligen behöver det har potentialen att skapa ett mycket effektivt arbetsflöde för programmeraren, som då kan kombinera både Pythons användarvänlighet och snabbheten av C. Fokus i denna rapport kommer främst att ligga på prestandaeffekterna av detta, då detta är Cythonprojektets huvudsakliga fokus, men även användarperspektivet och felsäkerheten kommer beröras, då detta är en integral aspekt av alla typsystem.

1.1 Problemformulering

Vilka fördelar, speciellt prestandavinster, tillför gradvis typning med Cython till Python?

Ordlista

API *Application Programming Interface* är regler för gränssnittet mellan olika program eller system, vilket beskriver hur de kan interagera med varandra.

CLR *Common Language Runtime* är körmiljön i Microsofts .NET-initiativ.

Dekorator Ett Pythonobjekt som kan kallas med inget eller ett argument, vilka modifierar funktioner eller metoder i en form av metaprogrammering. De kan sedan kallas genom syntaxen `@exempel`.

Dynamisk laddning av kod Dynamisk laddning av kod (ej att förväxla med dynamisk typning) innebär att ett program under körning kan ladda in bibliotek och köra dessa.

Funktionspolymorfi Funktionspolymorfi innebär att olika funktioner kan ha samma namn, så länge de har olika parametertyper.

Inferens är en metod för en kompilator eller interpretator att automatiskt bestämma typen av variabler i en källkod.

Mobil kod Kod vilken överförs mellan olika datorer eller system, och körs på mottagarsystemet utan explicit installation eller exekvering.

Overhead För att ett program ska utföra en viss uppgift behövs ibland kod som ej ingår i själva programmet, denna kod kallas för overhead.

Runtime-reflektion (eng.*reflection*) innebär att program under körtid kan observera och modifiera sin struktur, till exempel kontrollera typen på objekt och modifiera programmet därefter.

Undantag (Eng: Exception) är specialfall där programmets flöde ändras, används ofta för felhantering.

Kapitel 2

Teori och Bakgrund

2.1 Typsystem

En formell definition av ett typsystem, given av Pierce i *Types and programming languages*, är:

> ”A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”[22]

Fritt översatt:

> Ett typsystem är en lätthanterad syntaktisk metod för att påvisa avsaknaden av vissa programbeteenden genom att klassificera meningar i enlighet med de beräknade värdenas art.

Mer informellt kan man säga att ett typsystem är ett sätt för program och programmerare att tilldela mening till den binära data som datorn i grunden arbetar med. Genom att associera en typ med en bitsekvens kan program avgöra vilka egenskaper och operationer som datan stödjer. Genom att använda denna information kan typfel, eller felaktigt/oönskat beteende i ett program orsakat av diskrepanser mellan olika typer upptäckas.

Typsäkerhet är ett begrepp vilket betecknar att ett programmeringsspråks kompilator och/eller exekveringsmiljö kan upptäcka och förhindra många typfel.

Olika programmeringsspråk implementerar olika typsystem, vilka kan klassificeras enligt en rad måttstockar. Nedan presenteras de viktigaste för förståelsen av innehållet i denna rapport.

2.1.1 Stark typning

Stark typning innebär att typsystemet begränsar mängden operationer med flera ingående typer. Det finns ingen allmänt överenskommen exakt definition av begreppet, men egenskaper som brukar nämnas är:

Typsäkerhet: Operationer eller funktionsanrop med parametrar av ej stödda typer kan ej utföras, till exempel kan kompilatorn ej tillåta kompilering då denna typ av fel upptäcks. Ett annat sätt att garantera typsäkerhet är att under körning skapa ett väldefinierat fel; ofta i formen av ett *undantag*. Notera dock att svag typning inte automatiskt innebär typosäkerhet; se 2.1.2.

Fixa typer: Ett objekt kan inte byta typ under programmets körning, utom vid explicit typomvandling.

Avsaknad av sätt att kringgå typsystemet: Programmeraren skall inte ha direkt tillgång till typers underliggande bitsekvenser, då sådan tillgång medför att typsystemet kan kringgås.

Exempel på starkt typade språk

Python, Ruby, Java, C#, Haskell, Clojure

2.1.2 Svag typning

Svag typning är motsatsen till stark typning. Karakteristiskt för ett svagt typat språk är stöd för implicit typomvandling (omvandling mellan typer kan ske implicit), ad-hoc polymorfi (flera metoder med samma namn men med olika parametertyper stödjs) eller att programmeraren har tillgång till typers underliggande bitsekvenser. De mindre restriktiva reglerna för hantering av typer i ett svagt typat språk kan ge intrycket att svag typning är detsamma som typosäkerhet, men detta är inte nödvändigtvis sant; till exempel kan statisk typning eller andra begränsningar av vilken kod som är tillåten göra ett svagt typat språk typsäkert.

Exempel på svagt typade språk

Assembler, C, C++, Objective-C, Perl, Javascript.

2.1.3 Stark kontra Svag typning

Fördelar med svag typning kan vara att det ger programmeraren större möjligheter att skriva sin kod på valfritt sätt. Till exempel kan tillgång till underliggande bitsekvenser ibland användas för att skriva snabb kod.

Svag typning kan även minska mängden skrivande för programmerare. Till exempel: om kompilatorn gör implicita typomvandlingar slipper programmeraren skriva ut dessa explicit. Dessa extra möjligheter kommer dock till priset av minskad felsäkerhet och större krav på programmeraren; modifiering av de underliggande bitsekvenserna kan om programmeraren gör ett misstag leda till buggar. Implicita typomvandlingar riskerar att skapa problem om programmeraren inte avsåg att göra omvandlingen.

Kort sagt ger svagt typade språk en högre flexibilitet, men starkt typade språk löper å andra sidan en mindre risk för felaktiga programbeteenden orsakade av typdiskrepanser.

2.1.4 Statisk typning

Statisk typning är en begränsad form av programverifiering, och innebär att variabler associeras med en typ när programmet kompileras. Statisk typkontroll kräver oftast att typer stämmer överens i alla möjliga körningar av ett program. Typkontrollen tappar annars mycket av sin kraft, då den utan denna extra restriktion måste godkänna potentiellt typosäkra programbeteenden. Betrakta exemplet

```
1 if <komplext test> then
2     <fortsatt exekvering>
3 else
4     <typfel>
```

som i en statisk miljö kommer underkännas som otillåten kod, även om det komplexa testet alltid utvärderas till sant.[22]

Explicit kontra Implicit typning

Implicit och explicit typning är två olika sätt för kompilatorn att avgöra typen av olika värden under statisk typning. Explicit typning innebär att programmeraren explicit måste informera kompilatorn om olika variablers typer med hjälp av typannotationer. Implicit typning är motsatsen och innebär att kompilatorn själv deducerar variablers typer med hjälp av inferens. Implicit, statisk, säker typning begränsar antalet möjliga programbeteenden i ett språk, och är därför relativt ovanligt. Notera att de flesta statiska språk varken är 100% explicita eller 100% implicita utan befinner sig i en kontinuerlig skala däremellan.

Exempel på explicit statiskt typade språk

Java, C, C++, Objective-C

Exempel på implicit statiskt typade språk

Haskell, Scala

2.1.5 Dynamisk typning

Dynamisk typning är motsatsen till statisk typning och innebär att all eller en majoritet av typkontrollen sker under körning. I ett dynamiskt typsystem har värden typer och en variabel kan under ett programs körning innehålla flera olika typer. Detta typsystem innebär att typfel kan ske under körning, men detta betyder inte

att ett dynamiskt typat språk behöver vara typosäkert eller svagt typat. Det är till exempel vanligt att typfel under körning ger upphov till ett undantag.

Exempel på dynamiskt typade språk

Python, JavaScript, Lisp, PHP, Prolog

2.1.6 Duck Typing

Duck typing är en form av dynamisk typning där ett objekts metoder och/eller egenskaper avgör vilken semantik som är giltig, istället för objektets klass/superklass eller huruvida klassen implementerar ett gränssnitt.[31] Runtime-typkontrollen är alltså endast intresserad av vilka metoder eller fält ett objekt har, och inte dess ”sanna” typ. Även vissa i övrigt statiskt typade språk har stöd för Duck typing, genom att deklarera variabler som fria objekt utan förutbestämd klass varpå runtime-miljön testar huruvida den fria klassen implementerar sökta metoder.[3]

Det udda namnet på typsystemet kommer från ”ank-testet” ursprungligen myntat av James Whitcomb Riley: [14]

> ”When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck”

och syftar på att duck typing egentligen inte är ett sätt att hantera *typer* men att resultatet ändå har formen av ett typsystem.

Exempel på språk som använder eller stödjer Duck typing

Python, Objective-C, Perl, C#

2.1.7 Statisk vs Dynamisk typning

Fördelar med statisk typning

En fördel som burkar tillskrivas statiskt typade språk är tidig upptäckt av typfel, till exempel addition av en booelsk variabel med ett heltal.[18][2] Genom att programmeraren får information om felet inom ett kort tidsintervall från själva kodningen, kan felet vara lätt att lokalisera och åtgärda. I ett dynamiskt typat språk finns alltid risken att ett typfel förblir oupptäckt en längre tid vilket kan försvåra felets åtgärdande. Notera dock att ett dynamiskt typat språk även kan vara starkt och säkert typat (se 2.1.5); detta innebär här att ett typfel som blivit kvarlämnat i koden en tid efter programmeringen då det uppstår kan ge upphov till ett undantag vilket kan hanteras. Detta står i motsats till det odefinierade beteende ett svagt och dynamiskt typat språk kan uppvisa då ett typfel uppstår.

Statiskt typade språk med explicit typning ger programmeraren en verifierbar dokumentation i form av typdeklarationer. Dessa kan till exempel hjälpa programmeraren att bestämma vilka parametertyper respektive returtyper en metod har.

Denna information kan självklart även dokumenteras av programmeraren i en dynamiskt typad miljö som kommentarer i koden, men med dynamisk typning kan inte typkontrollsystemet verifiera att denna dokumentation stämmer överens med programmets faktiska implementation.

Statisk information kan även användas för att ge en bättre utvecklarupplevelse med hjälp av en integrerad utvecklingsmiljö med funktioner så som kodkomplettering eller menyer med metoder tillämpliga på ett specifikt objekt.[18]

Statisk information kan även ge prestandaförbättringar: Den extra typinformationen kan användas av kompilatorn till att göra optimeringar vid kompileringstid, till exempel genom att ersätta dynamiska metodanrop med statiska då metodmottagarens typ är känd. Effektiviteten kan förbättras än mer genom att använda faktumet att inte alla värden behöver vara associerade med en dynamisk typ. För att hålla typsäkerhet i ett dynamiskt språk måste värden ha en sådan dynamisk typ för att att kunna ge välformade fel då typdiskrepanser upptäcks.[18]

Fördelar med dynamisk typning

Vissa helt dynamiska programbeteenden kan inte kodas med statisk typning; detta gäller till exempel dynamisk laddning av kod, mobil kod, runtime-reflektion med flera.[18] I de fall statiskt typade språk behöver uppvisa sådant beteende måste man lokalt frångå det statiska typsystemet för ett dynamiskt vilket kan ge upphov till problem.[2]

Även mindre avancerade exempel finns där de hårdare restriktionerna i en statiskt typad miljö gör att rimliga programbeteenden ej tillåts. I 2.1.4 nämndes exemplet

```
1 if <komplext test> then
2     <fortsatt exekvering>
3 else
4     <typfel>
```

I en verklig applikation är det måhända lika bra att inte tillåta ful kod som denna. Men mer komplexa instanser av samma fenomen kan vara rimligt och till och med lämpligt programbeteende. Till exempel är det ibland rimligt att en metod returnerar olika typer beroende på någon parameter; ett programbeteende som ej tillåts under statisk typning. I de fall motsvarande beteende skall implementeras i en statisk miljö, måste man lösa det genom olika omvägar till exempel subklassning, vilket kan vara riskabelt då man frångår gängse beteende i en statiskt typad miljö.

Ett sista, och enklare argument för dynamisk typning vara att den extra arbetsbördan med att skriva ut typer i ett explicit, statiskt typat språk kan vara mer arbete än det är värt. Detta gäller i små projekt eller då en snabb prototyp är viktigare än felfrihet under exekvering, men även i inledningsfasen hos ett större projekt; särskilt då många olika typer skall skrivas ut i ett program under konstruktion, och programmet i en statiskt typad miljö inte kan kompileras utan en korrekt och full-

ständig typning given av programmeraren. Detta kan vara särskilt irriterande då många av typerna angivna för att få programmet att kompilera kan ändras i ett senare skede.[34]

Detta problem finns inte i statiskt, implicit typade språk till exempel Haskell, men dessa språk är på grund av sin okonventionella natur och branta inlärningskurva ofta inte ett praktiskt alternativ i tillämpade projekt.[32]

Sammanfattning

Statisk typning möjliggör att typfel kan upptäckas vid kompileringstid, och därmed eliminerar man risken som uppstår med dynamisk typning nämligen att typfel kan ske vid ett senare tillfälle, då de kan vara svårare att lokalisera och åtgärda. Statisk typning innebär även att kompilatorn kan utnyttja kunskapen om variablers typer till att optimera koden.

Dynamisk typning kan inte utnyttja dessa fördelar, men åtnjuter å andra sidan en större flexibilitet då dynamisk information och funktionalitet kan användas för att generera typer eller bestämma typen hos ett värde. Dynamiskt typade språk avlastar även programmeraren från att explicit deklarera typer i de fall då typen hos en variabel inte kan infereras.

2.1.8 Gradvis typning

Gradvis typning, även kallat *valfri typning*, är en kombination av statisk och dynamisk typning i samma språk. Gradvis typning är ett relativt nytt koncept jämte statisk eller dynamisk typning.[35] Idén är att utan typannotationer beter sig språket som om det vore dynamiskt typat. Programmeraren har dessutom möjligheten att i sitt program lägga till typannotationer; i närvaron av dessa antar variabler den statiska typen angiven av programmeraren. I teorin ger detta system fördelarna hos både dynamiska och statiskt typade språk, utan flera av deras respektive nackdelar.

Programmeraren kan snabbt göra en prototyp av sitt program med dynamisk typning, för att sedan gradvis lägga till statiska typannotationer om ett sådant behov finns; till exempel då programmet växer sig så stort att statiska typannotationer krävs ur ett underhålls- eller felsäkerhetsperspektiv, eller då programmet eller delar av det kräver den högre prestanda statisk typning kan ge.[34][21]

Gradvis typning är dock svårt att implementera, då statiskt och dynamiskt typade variabler och värden skall interagera med varandra. Det är också en utmaning att designa typkontrollen effektivt och att utnyttja optimeringspotentialen hos de statiska typerna.[34]

Exempel på gradvis typade språk

Cython, Clojure, C# (i viss mån)

2.2 Python

2.2.1 Introduktion till Python

Python är ett interpreterat högnivåspråk, vars grundfilosofi fokuserar på läsbarhet.[30] Python är objektorienterat med ett starkt, dynamiskt typsystem.[30] Liksom andra dynamiskt typade språk används Python ofta som ett skriptspråk; men det används även i en lång rad andra tillämpningar. Exempel är fullständiga applikationer, snabb utveckling av prototyper och program för vetenskapliga beräkningar.[34][20]

2.2.2 Exekveringsbeteende

Python är ur ett användarperspektiv interpreterat, men hur denna interpretation går till skiljer sig åt mellan olika implementationer av Python. Många av dessa implementationer har stöd för att använda Python som en kommandotolk, något som är otänkbart i ett kompilerande språk. Många implementationer är dock inte helt interpreterande i sitt sätt att behandla den Pythonkod som körs av interpretatorn. Referensimplementationen av Python, CPython, kompilerar vid körning av Pythonprogram dessa till bytekod som sedan körs i Pythons inbyggda virtuella maskin.[17, p.19][26][4]

En rad andra implementationer finns vid sidan av CPython, till exempel Jython vilken kompilerar Python till Java-bytekod för lättare integrering med Java och Javas virtuella maskin,[15] eller IronPython vilken istället kompilerar CLR-bytekod för interaktion med Microsofts .NET-ramverk.[19] Fokus i denna rapport ligger på Cython, ett superset till CPython vilken beskrivs i detalj i 2.3.

2.2.3 Moduler

En *modul* är den högsta enheten för programorganisation i Python. En modul fungerar som ett ”paket” med kod som kan importeras och köras av interpretatorn. Till många Pythonimplementationer medföljer ett standardbibliotek av moduler, men moduler kan även användas för att dela upp egenskriven kod i separata filer.[17, pp.247-256] Moduler skrivs oftast i Python men vissa implementationer tillåter även moduler skrivna i implementationsspråket, i Jython kan man till exempel skriva moduler i Java och i CPython i C, men dessa måste då implementera sina respektive API:er vilka har en hög inlärningskurva.[5]

2.2.4 Typsystem

Python är som nämndes ovan starkt (se 2.1.1) och dynamiskt (se 2.1.5) typat. Det dynamiska typsystemet är av typen *Duck typing* (se 2.1.6). Försöker programmeraren komma åt en metod eller attribut som inte finns, skapar interpretatorn ett undantag som måste fångas för att inte avbryta programexekvering.[23] En vanlig kodstil i Python, som även används i implementationen av CPython är kodstilen att programmet inte verifierar att metoder finns innan dessa kallas, utan istället

använder *try-except-block* för att fånga eventuella undantag orsakade av att metoder eller attribut inte finns.[38, p.27]

Det är möjligt att vid behov frångå Pythons duck typing genom att använda metoder som `type()` eller `isinstance()`, men detta är generellt sett mot Pythons stilguide.[38, p.26]

I de senaste versionerna av Python (version 3.0 och framåt) finns stöd för parameter- och returtypsannotationer i koden. Dessa används dock i nuläget inte av interpretatorn.[40] Dessa annotationer kan dock nås av programmet i runtime, och genom att använda dessa kan program eller moduler frångå den helt dynamiska typkontrollen till förmån för vad som närmast motsvarar statisk typning av parametrar och returtyper. De moduler som finns för detta ändamål ger dock inga prestandaökningar.[39]

2.3 Cython

2.3.1 Vad är Cython

Cython är ett programmeringsspråk som har sin grund i Python men med extra syntax för gradvis typning, direktanrop av C-funktioner med mera. Cython översätter Cython- eller Python-kod till C-kod vilken sedan kompileras till maskinkod. De kompilerade binärerna är snabbare än interpreterad eller bytekodad Python, och kan användas av CPython-interpretatorn som moduler eller direkt av operativsystemet som självständiga applikationer.[5]

Cython är i skrivandets stund på versionsnummer 0.14.1,[37] och alltså ett pågående projekt. Cythonprojektets slutgiltiga mål är att eliminera behovet att anropa CPythons C-APIer, utan att prestandan blir lidande, samt att bli en del av standarddistributionen av Python.[36] Detta skulle i sin tur skulle innebära att Python blir ett gradvis typat språk.

2.3.2 Funktionsanrop

När en funktion i ett program anropas måste parametrarna till denna på något sätt skickas med, detta kallas för parameterpassning. När en funktion i Python anropas finns ett visst overhead vilket innebär längre exekveringstid. Funktionsanrop i Python går till på följande sätt:[13][27]

- Den anropande koden måste skapa en Python-tupel för att hålla argumenten. En tupel i Python är en ordnad oföränderlig lista som precis som en vanlig lista kan hålla blandade typer av objekt i samma lista.

- Den anropande koden måste slå upp funktionen genom sitt namn (vid upprepade anrop sparas funktionen i en cache).

- Python-API används för att anropa funktionen.

- Den anropade funktionen måste i sin tur använda Python-API för att läsa argumenten från tupeln.

Det är bland annat den tid det tar att packa ned och upp tupeln med argument samt att göra funktionsuppslagning som är overhead, vilket gör funktionsanrop i Python långsammare än mer maskinnära språk.

Om en funktion deklareras med *cdef* i Cython kommer anrop till funktionen ske med parameterpassning likt C vilket innebär att parametrarna, beroende på vilken C-kompilator som används, skickas direkt via processor-register och/eller via stacken.

När en funktion har deklarerats med *cdef* kan den inte längre anropas från Python eftersom den översatts till en ren C-funktion. Cython erbjuder därför alternativet *cpdef*, vilken utöver den C-funktion som genereras av *cdef* även gör en Pythonfunktion som, om den anropas från Python, översätter Pythonargumenten till C och i sin tur kallar C-funktionen. Detta gör det möjligt att kalla funktionen från både en C-kontext och från Python.[6] När en funktion deklareras som en C-funktion finns även möjlighet att gradvis typa returtyper vilket, precis som med variabler, gör att typkontroll inte behöver köras i runtime. Notera att detta inte är sant om funktionen kallas från en Pythonkontext; då måste returvärdena ändå typkontrolleras och konverteras.

2.3.3 Klasser och attribut

I Python kallas objektvariabler för attribut och dessa är lagrade i en Pythontupel. Om en klass med hjälp av Cython deklareras *cdef* gör Cython dessutom om fälten i denna klass till en statisk C-struktur. Det finns då två olika sätt att nå klassens attribut; antingen genom Pythontupeln eller genom C-strukturen. Cythonkod använder C-strukturen för snabbare exekvering, men Python kan endast nå dem genom Pythontupeln.[7]

2.3.4 Inferens

Cython har ett begränsat inferenssystem, vilket automatiskt kan inferera typen hos variabler om de är av typ `double` eller `complex`. Detta beror på att matematiska operationer på dessa typer beter sig på samma sätt i Python och C.[33] Vid explicit typning av variabler av andra typer kommer matematiska operationer inte i alla lägen bete sig på ett exakt ekvivalent sätt som de gör i CPython. Till exempel kommer en variabler som explicit har typats till en `int` ha ett maxvärde på 2^{31}[8] medan CPython kan hantera värden av oändlig precision.[28][8]

Cythons typ-infererare har även möjlighet att inferera typer på variabler om de beror på andra variabler som är explicit typade.[12] Notera att de beroendena variablernas typer måste vara kända innan tilldelningen, se följande kod:

```
1 a = 5
2 cdef int b = 5
3 cdef int res = a + b
```

I denna kod kommer `a` inte infereras till int, utan istället betraktas som ett Pythonobjekt, vilket leder till att `b`, på rad 3, kommer omvandlas till ett Python-intobjekt, `a` och `b` adderas med anrop till Python C-API, och slutligen konverteras resultatet till en C-int igen för att kunna sparas i variabeln `res`. Detta tar mycket lite tid, men om inte `res` används till någonting vilket kan optimeras genom att ersätta Pythonmetodanrop med C-anrop har man förlorat ett litet mått prestanda istället för att vinna prestanda med den statiska typdeklarationen. På grund av detta är det ur prestandasynpunkt önskvärt att minimera mängden operationer där både dynamiskt och statiskt typade variabler ingår.

Om beroendet är känt innan tilldelning kan inferens ske och konvertering till Pythonobjekt behöver inte ske vid addition, se:

```
1 cdef int a = 5
2 cdef int b = 5
3 res = a + b
```

Här kommer alltså `res` automatiskt infereras till C-int vilket innebär att addition kan ske direkt i C och ingen konvertering till eller från Pythonobjekt behöver ske.

2.3.5 Översättning och kompileringsförfarande

Översättning och kompilering med Cython genererar antingen en modul vilken kan importeras till CPython, eller ett självständigt program. När Cythonkod har genomgått kompileringsfasen, det vill säga steget efter översättning till C-kod, är den låst till den arkitektur som den kompilerades för. Statiskt typkontroll av statiskt typade variabler sker i översättningsfasen, som inte kan fullföljas om Cython upptäcker ett typfel. Denna typkontroll är dock inte perfekt, och släpper ibland igenom typfel vilka sedan upptäcks i runtime, till exempel tillåter översättaren att heltal tilldelas till variabler som är explicit typade som Pythonsträngar.

Alternativt kan `Pyximport` användas. Denna Pythonmodul följer med Cython, och hanterar import och kompilering av Cython-filer automatiskt och dolt för användaren. På samma sätt som vanliga modulimporter i Python i det dolda kompilerar Pythonkod till bytekod så översätter `pyximport` den Cythonkod som importeras till C och kompilerar den till maskinkod.[9]

Fördelen med att använda *pyximport* är att koden inte längre är låst till den arkitektur den kompilerades för. Nackdelen är att det kräver att det exekverande systemet måste ha Cython installerat, samt att inga externa C-bibliotek kan användas. Att använda *pyximport* innebär också att Cythonkoden vid importtid måste översättas och kompileras, vilket innebär en tidsförlust.[24]

2.3.6 Exekveringshastighet Cython kontra Python

Cython syftar till att göra Python snabbare. Att översätta och kompilera Pythonkod i Cython ger i sig en prestandavinst, mest på grund av att koden kompileras till maskinkod istället för bytekod.

När Cython känner till typen av variabler genom explicita deklarationer eller inferens, och dessa har en ekvivalent typ i C, kan Cython hantera dessa som C-variabler. Operationer på variabler som alla är av C-typ kan använda C-funktioner istället för att gå via Pythons API. I program med många funktionsanrop ger också den parameterpassning som Cython erbjuder en minskad exekveringstid. Detta beror på det overhead som finns med hantering av argumenten som tupler.[6] Åtkomst till attribut blir också snabbare då man vid åtkomstförfrågan inte behöver hantera en Python-tupel utan kan använda snabbare C-strukturer.[7]

Förutom de ovan nämnda optimeringarna, gör Cython en stor mängd andra, mindre, optimeringar vilka faller utanför denna rapports omfattning.

2.3.7 Cythons begränsningar

Denna del är baserad på rapporten *Cython: The best of both worlds* av Stefan Behnel m.fl.

Jämfört med kodning i ren Python

Cythons huvudsakliga svagheter jämfört med kod skriven i ren Python är den extra kompileringstiden och minskad portabilitet, då översättningen av koden kräver att Cython är installerat på det översättande systemet. Detta kan lösas genom att använda Cythons *"pure mode"* i vilket så kallade *dekoratorer* används för att indikera typdeklarationer, i motsats till typdeklarationer direkt i koden. Detta gör att koden kan köras i ren Python; typdeklarationerna i dekoratorerna ignoreras i detta fall; men vid eventuell kompilering med hjälp av Cython används typdeklarationerna för optimering.[11] Att använda dessa är dock osmidigt jämfört med att skriva typdeklarationer direkt i koden.

Alternativt får de från Cython kompilerade C-källfilerna (ej binärerna) skickas med pythonkoden. Dessa kan sedan kompileras med valfri C-kompilator.[5]

En annan svaghet som tas upp i *Cython: The best of both worlds* är kravet på en separat bygg-fas.[5] Detta är dock inte helt sant, då modulen `pyximport` lanserad i Cython v.0.11 tar bort behovet av en sådan separat bygg-fas, i varje fall sett ur ett användarperspektiv; se 2.3.5.

Jämfört med ren C

Jämfört med C har Cython ett svagt stöd för parallellism med delat minne; detta på grund av Pythons (och därmed också Cythons) globala interpretatorlås (GIL) vilken blockerar kod som inte är trådsäker. Cython kan undvika detta genom att

deklarera sektioner som ren C-kod som då kan köras parallellt, men detta blir snabbt svårhanterligt.[5]

Meddelande-parallellism med hjälp av multipla processer åtnjuter å andra sidan ett bra stöd i Cython.[5]

Saknad funktionalitet

Cython strävar efter att vara ett superset av Python, men har inte riktigt nått dit i nuläget, då Cython saknar stöd för vissa funktioner och beter sig annorlunda i vissa fall. Till exempel stöds inte generator-uttryck, och referenser till klassmetoder i klassdeklarationer ger inte funktionsreferenser som i Python utan istället obundna metoder.[10] Den saknade funktionaliteten är relativt liten och skall försvinna i framtida versioner, men i nuläget utgör avsaknaden av dessa funktoner en risk då den i sällsynta fall kan skapa svårupptäckta fel.

Kapitel 3

Metod

3.1 Metodsammanfattning

För att testa Cythons beteende och prestandavinster vid gradvis typning utformades tre testfall:

- Rekursiv beräkning av fibonaccital (se 3.2.1)
- Runge-Kutta-metoden av fjärde ordningen (se 3.2.2)
- Filtrering i en nyckel-värde-datastruktur (se 3.2.3)

Dessa tre tester är valda för att belysa olika aspekter av optimering med gradvis typning, och är inte tänkta att vara representativa för typiska Pythonprogram.

Pythonkod skrevs som implementerade de tre testfallen. Dessa typades sedan statiskt i varierande grad med hjälp av Cythons system för gradvis typning. Dessa olika grader (nedan kallade versioner) finns beskrivna i 3.3.

Tidsåtgången för samtliga versioner av de tre testfallen mättes; tillvägagångssättet för denna tidsmätning beskrivs i 3.4. För diskussionen inspekterades även den av Cython genererade C-koden.

3.2 Testfall

3.2.1 Rekursiv fibonaccitalsberäkning

Motivering

Detta testfall inkluderades i undersökningen för att utreda vilka prestandaeffekter typdeklarationer i Cython ger i en metod som använder mycket rekursion och därmed också en stor mängd metodanrop relativt beräkningar i metodens kropp.

Notera att den naiva rekursiva metoden för att beräkna fibonaccital som används i denna undersökning är enormt ineffektiv jämfört med metoder vilka använder tabelluppslagning. De tillagda statiska typningarna i denna funktion skall alltså

inte ses som ett exempel på bra optimering, utan endast som ett exempel på hur Cythons valfria statiska typningar kan se ut och hur de påverkar prestandan.

Definition

Fibonaccital är tal definierade som

$$F_n = F_{n-1} + F_{n-2}$$

med

$$F_0 = 1$$
$$F_1 = 2$$

Implementation

Fibonaccitalen beräknas i den implementerade funktionen med en naiv rekursiv metod. Funktionen returnerar `fib(n-1)+fib(n-2)` i enlighet med den matematiska funktionen. Se koden i bilaga .

3.2.2 Runge-Kutta-metoden av fjärde ordningen

Motivering

Denna matematiska metod valdes för att den kräver addition, multiplikation och division under många iterationer vilket gör det till en matematiskt beräkningsintensiv funktion. En stor del av variablerna är flyttal, och kan behandlas som den maskinnära C-typen *float*.

Implementationen definierar även en metod med två returvärden, något som är specifikt för Python och inte finns i C.

Definition

Runge-Kutta-metoden av fjärde ordningen (nedan RK4) är en metod för att beräkna lösningar till ordinära differentialekvationer. Metoden löser så kallade begynnelsevärdesproblem där indata är en startpunkt och en derivata-funktion.
RK4 är en stegmetod som gör beräkningar för någon begynnelseparameter $t = t_0$ och sedan stegar fram med en fix steglängd h för att sedan upprepa beräkningarna med parametern $t_{i+1} = t_i + h$. Som steglängd kan ett godtyckligt tal användas; mindre steglängd medför fler iterationer och längre beräkningstid men ger ett noggrannare resultat.

Om ett begynnelsevärdesproblem är specificerat som

$$y' = f(t, y), \quad y(t_0) = y_0$$

så ger RK4

$$y_{n+1} = y_n + \tfrac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$
$$t_{n+1} = t_n + h$$

Där y_{n+1} är RK4-approximationen av $y(t_{n+1})$ och

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{aligned}$$

[16, pp.97-98]

Implementation

En funktion för att beräkna ett steg n anropas iterativt. Stegberäkningsfunktionen anropar deriveringsfunktionen $f(t, y)$ och utför beräkningarna för det aktuella steget. Funktionen som simuleras (y) valdes arbiträrt och modellerar positionen hos en harmoniskt svängande fjäder, och derivatan $(y' = f(t, y))$ motsvarar följaktligen fjäderns acceleration. Se koden i bilaga .

Steglängden h valdes arbiträrt till 0,001 och vid anrop av testfunktionen varieras simuleringslängden.

3.2.3 Dictionary-filter

Motivering

Detta testfall inkluderades för att testa Cythons prestandavinster vid deklarationer av typer som inte kan omvandlas till C-typer. Det är tänkt att agera som ett negativt test, då Cython eventuellt inte kan få samma prestandavinster vid införandet av statiska typdeklarationer av Python-objekt som vid statiska typdeklarationer av typer som kan konverteras till C-typer.

Implementation

Ett uppslagsverk (`Dict`) i Python är en inbyggd klass vilken implementerar en nyckel-värde-tabell med en hashtabell. Nycklarna är godtyckliga oföränderliga objekt, till exempel `String` eller `Int`, och olika nyckeltyper kan blandas i samma `Dict`. Värdena i en `Dict` kan, i dynamisk anda, innehålla objekt av olika godtyckliga typer. Nycklar måste vara unika; med andra ord kan inte en nyckel peka ut flera objekt. Python stödjer fler nyckel-värde-typer än `dict`, men inga andra sådana typer följer med standardimplementationen av Python.[25]

Vår testmetod `dict_filter` tar två parametrar, `dictionary` och `removeValue`. Returvärde är en `Dict` där alla nyckel-värde-par vars värde är `RemoveValue` har tagits bort. Detta implementeras genom att iterera genom alla nycklar och ta fram de associerade värdena. Om ett associerat värde skall filtreras bort görs detta med metoden `Dict.pop()`.

För att testa denna metod genereras en `Dict` av storlek N med alla heltal i intervallet $[1 - N]$ som nycklar. Till dessa nycklar associeras sedan slumpvis utvalda bokstäver, gemena eller versala, ur det engelska alfabetet. Sedan filtrerar vi bort

alla nycklar med gemener som värden genom att köra `dict_filter` med samma `Dict` och alla olika gemener. Tiden för denna iteration över alla 26 gemener är den tid som redovisas i resultatet.

Koden för `dict_filter` finns i bilaga A.3.

3.3 Testens utformning

Testfallen specificerade i 3.2 implementerades i vanlig Python. För att testa Cython och framförallt den gradvisa typningens inverkan på koden implementerades flera olika versioner av testfallen, utformade för att belysa skillnaden i exekveringstid med olika grader av typning.

3.3.1 Version interpreterad Python

För perspektiv på Cython jämfört med Python inkluderar vi originalimplementationen i ren Python, interpreterad av CPython; det vill säga kompilerad till Python-bytekod, se 2.2.2.

3.3.2 Version Cython-kompilering

För att mäta hur exekveringstiden påverkas av att översätta programmet till C med Cython och kompilera det gjorde vi just detta, utan att göra några ändringar i koden. Denna version är inte direkt relevant för mätningar av prestandaeffekten av gradvis typning, då inga typdeklarationer har införts. Det kan ändå vara intressant att se hur stor skillnad det är mellan körning i Pythons virtuella maskin och körning av det översatta C-programmet.

3.3.3 Version Cython med C-funktionsanrop

När man med Cython deklarerar en funktion med *cpdef* innebär det att denna kommer, om möjligt, kallas som en C-funktion. Som beskrivet i 2.3.2 innebär detta ofta en stor vinst i exekveringshastighet; men denna vinst grundar sig i pythons sätt att kalla metoder och inte direkt i användningen av statiskt typade variabler. De resterande versionerna kommer bygga på denna version. Denna version används också som referens för resultatet då resterande versioner inte kommer tillägga annat än just gradvis typning. Detta gör det möjligt att mäta den påverkan som gradvis typning har på exekveringshastigheten.

3.3.4 Version Cython med returtypdeklarationer

Denna version är ämnad att visa skillnaden med typdeklarerade returvärden jämfört med referensfallet. Denna version innebär ofta ett relativt få modifikationer i koden då antalet funktioner, och därmed också antalet returtyper jämfört med rader kod ofta är litet.

3.3.5 Version Cython med parametertypdeklarationer

Denna version är ämnad att visa skillnaden med typdeklarerade parametervärden jämfört med referensfallet. Denna version innebär, likt 3.3.4, få modifikationer i koden då antalet parametrar jämfört med rader kod ofta är litet.

3.3.6 Version Cython med retur- och parametertypdeklarationer

Denna version bygger vidare på version 3.3.4 och lägger, där möjligt, till deklarationer för parametrarna till funktionen. Precis som 3.3.4 innebär detta ofta liten påverkan på koden, av samma anledning som beskrivs där.

3.3.7 Version Cython med retur-, parameter- och variabeltypdeklarationer

Denna version är byggd för att visa vinsten av att typa så mycket som möjligt. Utformning av denna version innebär ofta många tillägg i koden då typdeklarationer görs på samtliga returvärden, parametrar och variabeldeklarationer. Programmet kommer i denna version vara fullständigt statiskt typat, bortsett från de variabler som inte kan typas på grund av deras dynamiska natur.

Endast RK4 använder variabler och är därför det enda testet som har denna version.

3.3.8 Version Cython med variabeltypdeklarationer

Som referens kommer även variabeltypdeklarationer utan parameter- och returtypsdeklarationer att testas. Detta för att se vilken påverkan endast variabeldeklarationer har på exekveringstiden.

Av samma anledning som i 3.3.7 finns denna version av testet bara för RK4.

3.4 Metod för tidsmätningar

För varje testfall skrevs ett Pythonskript vilket automatiskt testade alla olika versioner av testfallet i fråga. Varje version av testfallet testades med egen tidtagning för att påvisa skillnad i körningstid mellan de olika versionerna. Skriptet kördes även med flera olika indata för att se hur detta påverkade resultatet. De olika versionerna av samma testfall testades ”om lott” för varje indata, för att minimera påverkan av andra processer på datorn. Under testfallens körning lämnades testdatorn orörd med energisparläge samt skärmsläckare avaktiverade.

Tidsmätningsskriptet kördes interpreterat i CPython. På grund av detta måste alla anrop till testversioner definierade som C-funktioner från tidsmätningsskriptet gå genom en Pythonwrapper, se 2.3.2. Detta är tänkt att simulera ett verkligt användningsscenario, då modulerna konstruerade i Cython lämpar sig bäst för att användning från interpreterad Python.

Det spann av indata som testades valdes så att exekveringstiden för den snabbaste versionen hamnade ungefär mellan 1 ms och 5 s. Den undre gränsen 1 ms valdes för att undvika stor procentuell påverkan på resultatet från osäkerhet i tidsmätningsmetoden; konfidensintervallet för tidsmätningar av denna längd uppmättes experimentellt till $\pm 0,05$ ms vid 1% signifikansnivå. Den övre gränsen 5 s valdes arbiträrt för att hålla tiden för testerna på en hanterbar nivå, då de långsammaste programversionerna tar betydligt längre tid för samma indata.

För att köra de olika versionerna importerades dessa som moduler. Detta gjordes med Pythons `import` för Pythonversionen och med Cythons `pyximport` för de andra versionerna. Cython kördes i sitt standardutförande, det vill säga med alla kompilatordirektiv satta till standardvärden. För mätning användes Pythons modul `time` och dess funktion `clock()` som enligt Pythons dokumentation är den rekommenderade funktionen att använda för prestandatest.[29]

Tidsmätningen utfördes på en MacBook 6,1 med en Intel Core 2 duo-processor med dubbla kärnor på 2,26 GHz samt 6 GB ram.

Kapitel 4

Resultat

4.1 Hur resultaten presenteras

Här prestenteras data från tidsmätningarna av de olika versionerna specificerade i 3.2. Koden för originalimplemetationen i Python samt den mest typade versionen för varje testfall återfinns i bilaga B.

Medelvärdet för den procentuella prestandaförändringen jämfört med orginalversionen i ren Python redovisas direkt i denna del.

Rådatan från mätningarna i tabellform med exekveringstid och procentuell vinst i tid relativt referensversionen (specificerad i 3.3.3) återfinns i bilaga B. Diagram över exekveringstiden för varje testad indata och version redovisas i diagramform i bilaga C.

4.2 Rekursiv fibonccitalsberäkning

För beskrivning av testfallet se 3.2.1.

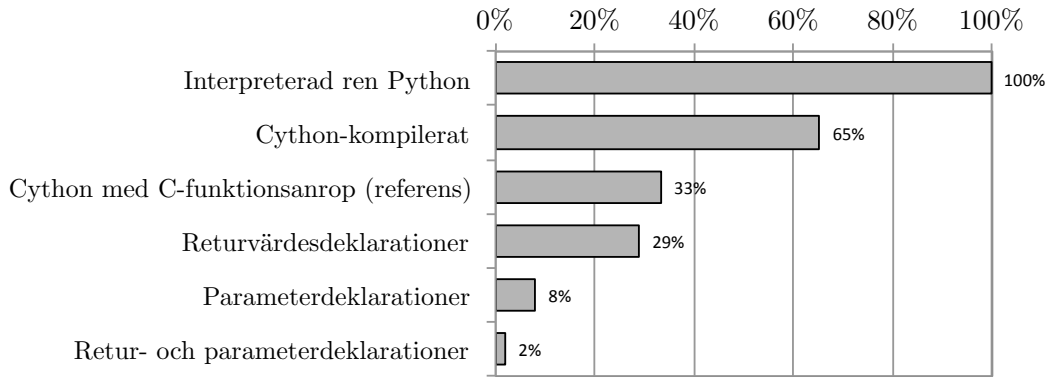
4.2.1 Implementerade versioner

I detta problem finns inga variabler att typdefiniera i funktionens kropp, varpå versionerna Cython med variabeldeklarationer och Cython med returvärdes-, parameter- och variabeldeklarationer ej återfinns i mätdata.

För koden till detta testfall, se bilaga A.1.

4.2.2 Tidsresultat

Figur 4.1. Tidsresultat för rekursiv fibonaccitalsberäkning



Diagrammet visar de procentuella hastighetsförbättringarna för respektive version i förhållande till orginalversionen i ren Python. Procenttalen är ett medelvärde beräknat över samtliga körningar.

Rådata från detta test finns i bilaga B.1, och diagram baserat på data i bilaga C.1.

4.3 RK4

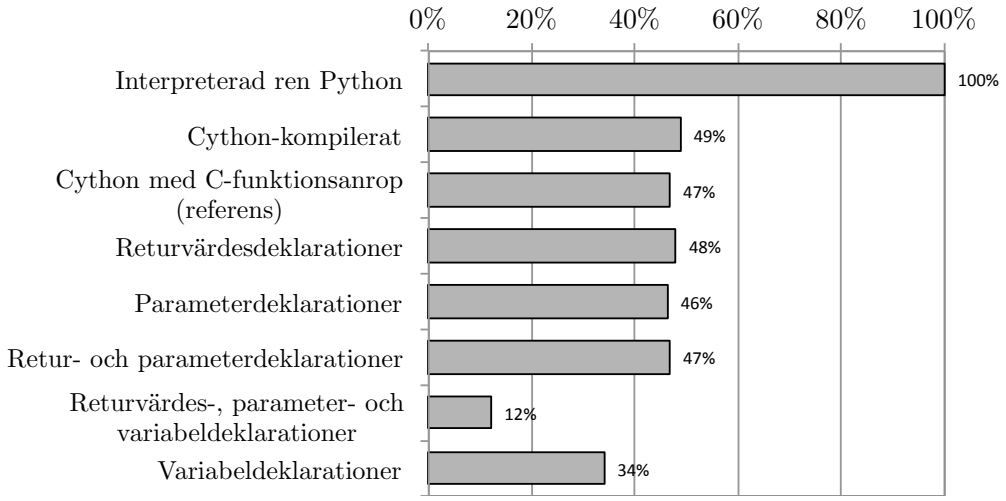
För beskrivning av testfallet se 3.2.2.

4.3.1 Implementerade versioner

Samtliga programversioner uppställda i 3.3 finns representerade för detta testfall. För koden till detta testfall, se bilaga A.2.

4.3.2 Tidsresultat

Figur 4.2. Tidsresultat för RK4



Rådata från detta test finns i bilaga B.2, och diagram baserat på denna data i bilaga C.2.

4.4 Dictionary-filter

För beskrivning av testfallet se 3.2.3.

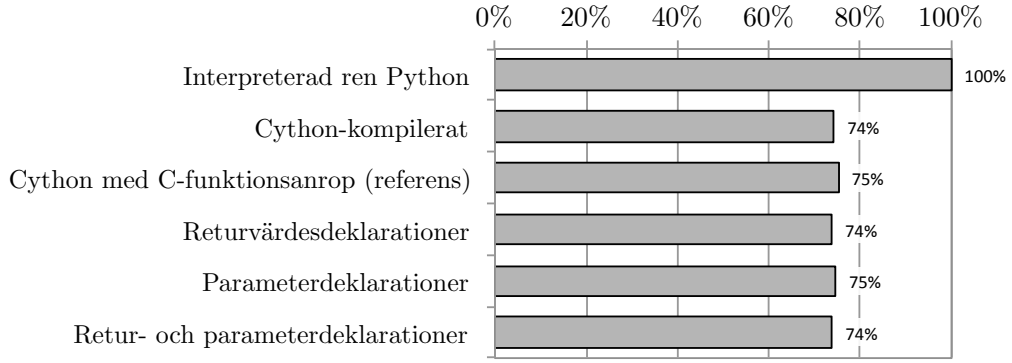
4.4.1 Implementerade versioner

På samma sätt som i 4.2 finns i implementation av testfunktionen inga variabler att typdefiniera inuti funktionens kropp, varpå versionerna med variabeldeklarationer ej återfinns i mätdata. Koden tillhörande detta testfall återfinns i bilaga A.3.

Notera att nyckel-värde-klassen typdefinierades som en `dict`, Pythons inbyggda nyckel-värde-klass. Detta gör att användning av denna funktion med en nyckel-värde klass av annan typ än `dict` genererar ett typfels-undantag under runtime, även om denna klass stödjer de använda metoderna. Detta står i kontrast mot orginalversionen i ren Python, där detta fungerar tack vare duck-typing. Då Cython ej stödjer funktionspolymorfi kan detta problem ej lösas genom multipla metoder.

4.4.2 Tidsresultat

Figur 4.3. Tidsresultat för Dictionary-filter



Rådata från detta test finns i bilaga B.3, och diagram baserat på denna data i bilaga C.3.

Kapitel 5

Diskussion

5.1 Rekursiv fibonaccitalsberäkning

Som tydligt syns på resultatet är detta testfall implementerat med en algoritm med exponentiell komplexitet. De olika Cython-optimeringarna förbättrar tiden för exekvering med konstanta faktorer.

5.1.1 Hastighetsförbättringar med översättning och C-funktionsdeklaration

Att med Cython översätta originalimplementationen till C för kompilering till maskinkod gav stora förbättringar i körtid. Att med Cython definiera funktionen som en C-funktion med kommandot `cpdef` gav ännu större förbättringar, i detta fall på grund av att rekursionen i funktionen kan kalla nästa rekursionssteg direkt i C-kod utan att gå via en Python-wrapper.

5.1.2 Hastighetsförbättringar med Gradvis typning

Returvärdestypdeklaration

Att deklarera returvärdet som `int` gav en liten prestandavinst jämfört med referensimplementationen. Detta beror troligtvis på optimeringar för kodraden
`return Fibonacci(n - 1) + Fibonacci(n - 2)`
där Cython kan kalla additionen med den snabbare metoden för C-heltal istället för att använda Pythons långsammare metoder för heltalsoperationer.

Parametertypdeklaration

Att deklarera parametern `n` som `int` gav en stor prestandavinst. Detta beror på att:

- if-satsen kan optimeras till en switch-sats i C på `n` eftersom `n` har typdeklarerats som en C-`Int`

- Subtraktionerna i satsen
  `return Fibonacci(n - 1) + Fibonacci(n - 2)`
  kan köras direkt med C-heltalsmetoder.

Retur- och parametertypdeklarationer

Typdeklarationer av både returvärde och parameter gav en mycket stor prestandavinst. I denna version är alla variabler statiskt typade som C-typer, och Cython kan då använda samtliga optimeringar nämnda i styckena ovan, och behöver utöver detta inte omvandla variabler till Pythonobjekt för vare sig funktionsanrop eller för retur av det beräknade värdet.

5.2 Runge-Kutta-metoden av fjärde ordningen

5.2.1 Hastighetsförbättringar med Gradvis typning

Returvärdesdeklarationer

Att endast lägga till returvärdesdeklarationer till koden innebar en liten försämring (ca -2%) av exekveringshastigheten. Eftersom den metod som itereras, `rk4`, returnerar en tupel, vilket är en typ som inte finns definierad i C, kan ingen optimering göras. Den marginella försämringen kunde inte förklaras.

Parameterdeklarationer

På resultaten från denna version kunde en liten prestandavinst observeras (ca 1%). Anledningen till denna vinst är dock ej känd.

Returvärde- och parameterdeklarationer

I versionen med retur- och parameter-deklarationer uppmättes ingen prestandaförändring. Detta tros beror på att variablerna i funktionen ej är statiskt typade vilket innebär att Pythons funktioner för matematik därför kommer användas, precis som i referensfallet.

Komplett typning

Samtliga matematiska beräkningar kunde göras av C istället för att anropa Pythons C-API, vilket tros vara anledningen till den prestandaförbättring som ses i resultaten. Att paramterarna är typade innebär också att ingen konvertering från Pythonobjekt till C-typer behövs göras i runtime vilket bidrar till den minskade exekveringstiden.

Variabeldeklarationer

Denna version gav också stor vinst relativt referensversionen. Detta av samma anledning som den komplett typade; de matematiska operationerna kunde i många fall utföras av C istället för av Python. Däremot kunde C-operationer ej utföras på värden som returnerades av den Pythonfunktion som definierade derivatan, vilken innebar att alla operationer med det värdet var tvungna att ske i Python.

5.3 Dict-filter

Dict-filter har linjär komplexitet, och hastigheten förbättras vid optimering med en konstant faktor.

5.3.1 Hastighetsförbättringar med översättning och C-funktionsdeklaration

Översättning till C ökade hastigheten för exekvering med ca 20%. Att deklarera funktionen som en C-funktion med `cpdef` gav marginellt mindre prestandavinst jämfört med ren Python; cirka en procentenhet sämre än versionen med endast översättning till C-kod. Detta är troligtvis på grund av den extra exekveringstid som spenderas i Python-wrappern vilken kallar C-funktionen, vilken inte ger något tillbaka då Dict-filter aldrig kallar sig själv eller kallas från annan C-kod.

5.3.2 Hastighetsförbättringar med Gradvis typning

Returvärdestypdeklaration till `dict` gav en marginell minskning i körtid, ca 2% jämfört med referensversionen. Typdeklaration av parametertypen till `dict` gav även den en marginell minskning i körtid, ca 1% jämfört med referensversionen. Typdeklarationer av både retur och parametervärden gav också den en marginell minskning i körtid, ca 2% jämfört med referensversionen. Dessa små skillnader är statistiskt signifikanta, men hur de uppstår kunde inte utrönas genom inspektion av C-koden översatt av Cython.

5.4 Sammantaget

5.4.1 Hastighetsförbättringar med översättning och C-funktionsdeklaration

Kompilering av Pythonkod med Cython gav i samtliga testfall presenterade i denna rapport en inte obetydligt reducerad körtid jämfört med CPython. Denna prestandaförbättring kan tillskrivas den snabbare exekveringen av maskinkod jämfört med bytekod i en virtuell maskin.

Att deklarera funktioner eller metoder som C-funktioner med Cython ger stora prestandavinster om det sker många anrop från C-kod till funktioner deklarerade

`cpdef`, eller då tiden spenderad i overhead för funktionsanrop är stor jämfört med tiden i själva funktionen. Fibonacci-testfallet i denna undersökning uppfyller detta, och mycket riktigt minskade exekveringshastigheten markant. I de andra testfallen är tiden spenderad i funktionsanrops-overhead låg och exekveringshastigheten påverkades endast marginellt av att deklarera metoder som C-funktioner.

5.4.2 Hastighetsförbättringar med Gradvis typning

Gradvis typning med Cython ger prestandaförändringar i spannet marginella prestandaförsämringar till mycket stora prestandavinster.

De marginella prestandaförsämringarna orsakas ofta av den extra typkontroll som den av Cython översatta C-koden måste göra.

En annan faktor vilken kan ha marginell negativ påverkan på prestandan är många omvandlingar mellan Python och C-typer, orsakat av att typinferensen i Cython endast infererar ekvivalenta typer (se 2.3.4)

Prestandavinsterna blir stora om många variabler typdeklareras som C-typer istället för Pythonobjekt, och om Pythonmetodanrop på eller med dessa typer kan ersättas med C-metodanrop som en följd av den extra typinformationen. Testfallet RK4 har många sådana variabler, och fick därför den största prestandavinsten vid många statiskt typade variabler. Typdeklarationer av Python-klasser ger endast marginella prestandaförbättringar, vilket illustreras av testfallet Dict-filter.

5.4.3 Användarperspektiv

Cythons gradvis typade typsystem fungerar väl givet sitt uttalade syfte som främst ett verktyg för optimeringar. I en majoritet av fallen upptäcks statiska typfel vid översättning; en av de största fördelarna med statisk typning.

Dessvärre kan ibland, även då samtliga variabler är typdeklarerade, inkonsistent typade program slippa igenom typkontrollen vid översättning. Till exempel tillåter översättaren tilldelning av flyttal till en variabel statiskt typad som en Pythonsträng. Detta beror på att funktionalitet för typkontroll för vissa av Pythons inbyggda typer ännu saknas men detta är något som utvecklarna bakom Cython avser implementera i framtiden.[1] De program som utför illegala operationer på icke-stödda Pythontyper ger istället välformade typfel i runtime då faktiska typer inte stämmer överens med de statiskt deklarerade. Detta kontrasterar mot ren Python, där felen inte hade upptäckts innan ett ogiltigt metodanrop på eller med typen hade gjorts.

Osäkerheten som uppstår på grund av detta förtar till viss del de fördelaktiga mjukvarukonstruktionsaspekter som statiskt typade variabler tillför, då programmeraren inte helt kan förlita sig på kompilatorn för att undvika typfel. Det är möjligt att framtida versioner av Cython delvis löser detta problem. Cython kan dock aldrig göra gradvis typad Pythonkod lika typfelsäker som kod i statiskt typade språk där alla möjliga programbeteenden kontrolleras för typfel, ty om så var fallet hade det gjort giltiga Pythonprogram ogiltiga i Cython.

Detta illustrerar den friktion som uppstår då dynamiskt typade värden skall interagera med statiskt typade variabler i ett gradvis typat system.

5.5 Felkällor

Tidmätningsmetoden som användes för mätningar har mycket hög precision, se 3.4.

Det är möjligt att processorarkitektur-specifika optimeringar hos kompilatorn kan påverka resultatets generalitet. Detta tros dock inte ha någon avgörande påverkan på resultatet eftersom samtliga test kördes på samma dator. Optimeringar gjorda av C-kompilatorer antas dock ligga på likvärdiga nivåer oavsett arkitektur.

Det största problemet för resultatens generalitet är utan tvekan den ringa mängden testfall; denna enskilda felkälla är så stor att de andra felkällorna kan betraktas som insignifikanta. En större mängd testfall hade varit högst önskvärt, men även större, mer realistiska testfall hade givit viktig data. Desvärre omöjliggjorde tidsbegränsningar för studiens utförande detta. Istället gjordes valet att fokusera på djupare analys av både tidsmätningarna samt optimeringsprocessen i sig hos få och små testfall. Inhämtad data och kunskap från denna analys användes sedan till att generalisera resultatet.

Kapitel 6

Slutsats

Gradvis typning med Cython är ett utmärkt sätt att optimera vissa Pythonprogram, framförallt beräkningsintensiva sådana och i synnerhet de vilka utför många matematiska beräkningar. I vissa synnerligen gynnsamma fall kan tidsåtgången sänkas till en nivå liknande den man kan förvänta sig av ett maskinnära språk så som C. Den förbättrade prestandan kommer dock till priset av minskad kodportabilitet och ibland mer svårläst kod.

Cythons gradvis typade typsystem fungerar bra, men är långt ifrån perfekt ur ett felsäkerhetsperspektiv sett. Cython är inte ett fullgott alternativ till ett statiskt typat språk, då det saknar flera av de fördelaktiga mjukvarukonstruktionsaspekter som gör statiskt typade språk till bra alternativ vid större projekt.

Tillägg till Cython så som mer intelligent inferenssystem, grundligare typkontroll samt bättre integrering med CPython, vilka finns som framtida mål för Cythonprojektet, skulle kunna minska omfattningen hos problemen ovan, och göra Cython till ett ännu mer attraktivt val i framtida versioner.

Litteraturförteckning

[1] E-postkorrespondens med R. Bradshaw och S. Behnel. URL: http://groups.google.com/group/cython-users/browse_thread/thread/bf3224100e3538a6.

[2] M. Abadi, L. Cardelli, B. Pierce och G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13:237–268, April 1991.

[3] Apple Inc. Mac OS X developer library: Objective C Objects, Classes, and Messaging, 2011-03-24. URL: http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocObjectsClasses.html#//apple_ref/doc/uid/TP30001163-CH11-SW1.

[4] Y. Asher och N. Rotem. The effect of unrolling and inlining for Python bytecode optimizations. *Proceedings of SYSTOR*, 2009.

[5] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn och K. Smith. Cython: The Best of Both Worlds. *Computing in Science Engineering*, PP(99):1, 2010.

[6] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein och G. Gellner. Cython Documentation: Early Binding for Speed, 2011-03-24. URL: http://docs.cython.org/src/userguide/early_binding_for_speed.html.

[7] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein och G. Gellner. Cython Documentation: Extension Types, 2011-04-05. URL: http://docs.cython.org/src/userguide/extension_types.html.

[8] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein och G. Gellner. Cython Documentation: Language Basics, 2011-04-12. URL: http://docs.cython.org/src/userguide/language_basics.html#differences-between-c-and-cython-expressions.

[9] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein, G. Gellner m.fl. Cython v0.14.1 documentation, 2011-03-31. URL: http://docs.cython.org/src/userguide/tutorial.html.

[10] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein, G. Gellner m.fl. Cython v0.14.1 documentation: Limitations, 2011-04-14. URL: `http://docs.cython.org/src/userguide/limitations.html`.

[11] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein, G. Gellner m.fl. Cython v.0.15pre Documentation: Pure Python Mode, 2011-03-31. URL: `http://readthedocs.org/docs/cython/en/latest/src/tutorial/pure.html`.

[12] R. Bradshaw. Enhancement proposal: Basic Type Inference, 2011-04-11. URL: `http://wiki.cython.org/enhancements/typeinference`.

[13] R. Bradshaw. Writing Fast Pyrex Code, 2011-04-05. URL: `http://wiki.sagemath.org/WritingFastPyrexCode`.

[14] R. Davis. *Who's Sitting on Your Nest Egg?* Bridgeway Books, City, 2007.

[15] B. Dehora. Jython Documentation, 2011-03-24. URL: `http://wiki.python.org/jython/WhyJython`.

[16] G. Eriksson. *NUMERISKA METODER med MATLAB*. NADA, KTH, 2002.

[17] M. Lutz och D. Ascher. *Learning python*. O'Reilly Media, 2004.

[18] E. Meijer och P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. I: *Proceedings OOPSLA Workshop On The Revival Of Dynamic Languages*. Citeseer, 2004.

[19] Microsoft Corporation. IronPython Documentation, 2011-03-24. URL: `http://ironpython.net/documentation/dotnet/`.

[20] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9:10–20, 2007.

[21] F. Ortin, D. Zapico, J. Perez-Schofield och M. Garcia. Including both static and dynamic typing in the same programming language. *Software, IET*, 4(4):268 –282, 2010.

[22] B. Pierce. *Types and programming languages*. The MIT Press, 2002.

[23] M. Pilgrim. Dive Into Python, 2011-03-25. URL: `http://diveintopython.org/`.

[24] P. Prescod. Pyximport, 2011-04-06. URL: `http://www.prescod.net/pyximport/`.

[25] Python Software Foundation. Python Documentation: Built in types: Dict, 2011-04-06. URL: `http://docs.python.org/library/stdtypes.html#mapping-types-dict`.

[26] Python Software Foundation. Python Documentation: Disassembler for Python bytecode, 2011-03-24. URL: `http://docs.python.org/library/dis.html#bytecodes`.

[27] Python Software Foundation. Python Documentation: Extending Python with C or C++, 2011-04-06. URL: `http://docs.python.org/extending/extending.html`.

[28] Python Software Foundation. Python Documentation: Numeric Types, 2011-04-14. URL: `http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex`.

[29] Python Software Foundation. Python Documentation: Time access and conversions, 2011-04-05. URL: `http://docs.python.org/library/time.html#time.clock`.

[30] Python Software Foundation. What is Python? Executive Summary, 2011-02-08. URL: `http://www.python.org/doc/essays/blurb/`.

[31] S. Rozsnyai, J. Schiefer och A. Schatten. Concepts and models for typing events for event-based systems. I: *ACM International Conference Proceeding Series*, band 233, ss 62–70, 2007.

[32] C. J. Sampson. Experience report: Haskell in the 'real world': writing a commercial application in a lazy functional lanuage. *SIGPLAN Not.*, 44:185–190, August 2009.

[33] D. S. Seljebotn. Cython trac: Always type-infer floating point types. URL: `http://trac.cython.org/cython_trac/ticket/460`.

[34] J. G. Siek och M. Vachharajani. Gradual typing with unification-based inference. I: *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, ss 7:1–7:12, New York, NY, USA, 2008. ACM.

[35] J. Siek och W. Taha. Gradual typing for functional languages. I: *Scheme and Functional Programming Workshop*, ss 81–92. Citeseer, 2006.

[36] W. Stein. Cython: C-Extensions for Python, Project Goals, 2011-04-05. URL: `http://wiki.cython.org/`.

[37] S. Tygier, R. Bradshaw, A. de Vore, D. S. Seljebotn och S. Behnel. Cython wiki: Relase history, 2011-04-06. URL: `http://wiki.cython.org/ReleaseHistory`.

[38] G. van Rossum. Using Python Release 2.6, 2011-03-17. URL: `http://users.info.unicaen.fr/~karczma/matrs/PyObj/Doc/docs-pdf/using.pdf`.

[39] C. Winter och I. Lowe. Type-checking module for Python, 2011-04-14. URL: `http://pypi.python.org/pypi/typecheck/0.3.1`.

[40] C. Winter och T. Lownds. PEP 3107 – Function Annotations, 2011-03-23. URL: http://www.python.org/dev/peps/pep-3107/.

Bilaga A

Kod

A.1 Fibonacci

A.1.1 Fibonacci orginalversion (dynamisk typad)

```
1 def Fibonacci(n):
2     if n == 0 or n == 1:
3         return n
4     else:
5         return Fibonacci(n - 1) + Fibonacci(n - 2)
```

A.1.2 Fibonacci med maximalt antal statiska typannotationer

```
1 cpdef int Fibonacci(int n):
2     if n == 0 or n == 1:
3         return n
4     else:
5         return Fibonacci(n - 1) + Fibonacci(n - 2)
```

A.2 RK4

A.2.1 RK4 orginalversion (dynamisk typad)

```
1  def rk4(x, v, a, dt):
2    x1 = x
3    v1 = v
4    a1 = a(x1, v1, 0)
5
6    x2 = x + 0.5*v1*dt
7    v2 = v + 0.5*a1*dt
8    a2 = a(x2, v2, dt/2.0)
9
10   x3 = x + 0.5*v2*dt
11   v3 = v + 0.5*a2*dt
12   a3 = a(x3, v3, dt/2.0)
13
14   x4 = x + v3*dt
15   v4 = v + a3*dt
16   a4 = a(x4, v4, dt)
17
18   xf = x + (dt/6.0)*(v1 + 2*v2 + 2*v3 + v4)
19   vf = v + (dt/6.0)*(a1 + 2*a2 + 2*a3 + a4)
20
21   return (xf, vf)
22
23 def accel(x, v, dt):
24    stiffness = 1
25    damping = -0.005
26    return -stiffness*x - damping*v
27
28 def runTest(h,stop):
29    t = 0
30    dt = h
31    state = 50, 5
32    euler = 50, 5
33
34    retstr = """Initial
35 -position: %6.2f,
36 velocity: %6.2f
37 """%state
38
39    while t < stop:
40      t += dt
41      state = rk4(state[0], state[1], accel, dt)
42
43    return retstr+"""
44 Final RK4
45 -position: %6.2f,
46 velocity: %6.2f"""%state
```

A.2.2 RK4 med maximalt antal statiska typannotationer

```
cpdef tuple rk4(double x, double v, a, double dt):
  cdef double x1 = x
  cdef double v1 = v
  cdef double a1 = a(x1, v1, 0)

  cdef double x2 = x + 0.5*v1*dt
  cdef double v2 = v + 0.5*a1*dt
  cdef double a2 = a(x2, v2, dt/2.0)

  cdef double x3 = x + 0.5*v2*dt
  cdef double v3 = v + 0.5*a2*dt
  cdef double a3 = a(x3, v3, dt/2.0)

  cdef double x4 = x + v3*dt
  cdef double v4 = v + a3*dt
  cdef double a4 = a(x4, v4, dt)

  cdef double xf = x + (dt/6.0)*(v1 + 2*v2 + 2*v3 + v4)
  cdef double vf = v + (dt/6.0)*(a1 + 2*a2 + 2*a3 + a4)

  return (xf, vf)

cpdef double accel(double x, double v, double dt):
  cdef double stiffness = 1
  cdef double damping = -0.005
  return -stiffness*x - damping*v

cpdef str runTest(double h, double stop):
  cdef double t = 0
  cdef double dt = h
  cdef tuple state = (50, 5)
  cdef tuple euler = (50, 5)

  cdef str retstr = """Initial
-position: %6.2f,
velocity: %6.2f
"""%state

  while t < stop:
    t += dt
    state = rk4(state[0], state[1], accel, dt)

  return retstr+"""
Final RK4
-position: %6.2f,
velocity: %6.2f"""%state
```

A.3 Dict filter

A.3.1 Dict filter orginalversion (dynamisk typad)

```
1 def filter(dictionary, removeValue):
2         for key in dictionary.keys():
3                 if removeValue in dictionary[key]:
4                         dictionary.pop(key)
5         return dictionary
```

A.3.2 Dict filter med maximalt antal statiska typannotationer

```
1 cpdef dict filter(dict dictionary, removeValue):
2         for key in dictionary.keys():
3                 if removeValue in dictionary[key]:
4                         dictionary.pop(key)
5         return dictionary
```

Bilaga B

Mätdata

B.1.1 Fibonacci mätdata

Indata	Körtid per version (s)					
Fibonaccital att räkna ut	Interpreterad ren Python	Cython-kompilerat	Cython med C-funktionsanrop (REFRERENS)	Returvärdes-deklarationer	Parameter-deklarationer	Retur- och parameter-deklarationer
fib(24)	0,0423	0,0275	0,0144	0,0128	0,0034	0,0008
fib(25)	0,0673	0,0439	0,0233	0,0204	0,0054	0,0014
fib(26)	0,1090	0,0704	0,0378	0,0327	0,0088	0,0022
fib(27)	0,1773	0,1139	0,0606	0,0526	0,0140	0,0036
fib(28)	0,2833	0,1845	0,0973	0,0845	0,0225	0,0058
fib(29)	0,4588	0,2983	0,1568	0,1356	0,0368	0,0094
fib(30)	0,7503	0,4819	0,2533	0,2180	0,0602	0,0151
fib(31)	1,2079	0,7798	0,4077	0,3511	0,0976	0,0244
fib(32)	1,9564	1,2647	0,6580	0,5659	0,1585	0,0398
fib(33)	3,1489	2,0538	1,0612	0,9149	0,2526	0,0640
fib(34)	5,1184	3,3309	1,7157	1,4782	0,4124	0,1034
fib(35)	8,3589	5,4361	2,7762	2,3909	0,6757	0,1674
fib(36)	13,6962	8,8760	4,4889	3,8689	1,1153	0,2709
fib(37)	22,4748	14,5076	7,2520	6,2693	1,7917	0,4382
fib(38)	36,3179	23,7279	11,7068	10,1594	2,9318	0,7094
fib(39)	58,9879	38,7756	18,9103	16,4744	4,7289	1,1473
fib(40)	95,2749	63,2169	30,5363	26,6913	7,7301	1,8564
fib(41)	154,0423	102,8356	49,3432	43,2027	12,4701	3,0037
fib(42)	245,9336	166,9335	79,8388	69,8578	20,0472	4,8604

B.1.2 Fibonacci procentuella körtider

Indata		version/interpreterad		version/referens		
Fibonaccital att räkna ut		Cython-kompilerat	Cython med C-funktionsanrop (REFERENS)	Returvärdes-deklarationer	Parameter-deklarationer	Retur- och parameter-deklarationer
fib(24)		65,08%	34,09%	88,89%	23,39%	5,86%
fib(25)		65,28%	34,65%	87,64%	23,18%	5,85%
fib(26)		64,57%	34,64%	86,71%	23,25%	5,84%
fib(27)		64,24%	34,17%	86,83%	23,16%	5,89%
fib(28)		65,13%	34,34%	86,85%	23,10%	5,93%
fib(29)		65,01%	34,18%	86,50%	23,47%	5,97%
fib(30)		64,22%	33,76%	86,07%	23,79%	5,96%
fib(31)		64,56%	33,75%	86,13%	23,95%	5,99%
fib(32)		64,64%	33,63%	86,01%	24,08%	6,05%
fib(33)		65,22%	33,70%	86,21%	23,81%	6,03%
fib(34)		65,08%	33,52%	86,16%	24,04%	6,03%
fib(35)		65,03%	33,21%	86,12%	24,34%	6,03%
fib(36)		64,81%	32,77%	86,19%	24,85%	6,03%
fib(37)		64,55%	32,27%	86,45%	24,71%	6,04%
fib(38)		65,33%	32,23%	86,78%	25,04%	6,06%
fib(39)		65,73%	32,06%	87,12%	25,01%	6,07%
fib(40)		66,35%	32,05%	87,41%	25,31%	6,08%
fib(41)		66,76%	32,03%	87,56%	25,27%	6,09%
fib(42)		67,88%	32,46%	87,50%	25,11%	6,09%
Medelvärde:		65,24%	33,34%	86,80%	24,15%	5,99%

B.2.1 RK4 mätdata

Indata Körtid per version (s)

Simulerad tid (simulearade sekunder)	Interpreterad ren Python	Cython-kompilerat	Cython med C-funktionsanrop (REFRERENS)	Returvärdes-deklarationer	Parameter-deklarationer	Retur- och parameter-deklarationer	Returvärdes-, parameter- och variabel-deklarationer	Variabel-deklarationer
2,0	0,0102	0,0049	0,0048	0,0048	0,0047	0,0047	0,0013	0,0035
185,6	0,9340	0,4609	0,4401	0,4563	0,4361	0,4370	0,1139	0,3193
369,3	1,8558	0,9079	0,8673	0,8806	0,8565	0,8678	0,2179	0,6240
552,9	2,7829	1,3730	1,2509	1,3270	1,2846	1,2996	0,3399	0,9368
736,5	3,7193	1,7975	1,7431	1,7810	1,7100	1,7044	0,4425	1,2247
920,2	4,5767	2,2608	2,1631	2,2060	2,1571	2,1617	0,5695	1,6039
1103,8	5,5351	2,7337	2,6002	2,6641	2,5692	2,5930	0,6525	1,9436
1287,4	6,4727	3,1666	3,0312	3,0734	3,0217	3,0169	0,7616	2,2352
1471,1	7,3424	3,5678	3,3933	3,5093	3,4269	3,4541	0,9094	2,5606
1654,7	8,2863	4,0997	3,9084	3,9597	3,8562	3,8948	1,0267	2,6954
1838,3	9,2225	4,4772	4,2865	4,4405	4,2733	4,3218	1,1232	3,2662
2022,0	10,2061	4,9578	4,7653	4,8641	4,7051	4,7236	1,1801	3,3179
2205,6	11,0999	5,3530	5,2273	5,2631	5,0954	5,1365	1,3889	3,7444
2389,2	12,0353	5,8342	5,5397	5,7104	5,5028	5,5797	1,4994	4,2650
2572,9	12,8266	6,2512	6,0707	6,1993	5,9776	6,0461	1,5757	4,3350
2756,5	13,7428	6,6974	6,4262	6,5250	6,3442	6,4487	1,6143	4,8737
2940,1	14,7310	7,2637	6,8761	7,0709	6,8254	6,8591	1,7603	4,8569
3123,8	15,7120	7,4033	7,3198	7,4697	7,2583	7,3321	1,8753	5,4560
3307,4	16,6629	8,0952	7,7909	7,9967	7,6760	7,7529	2,0441	5,7544
3491,0	17,4510	8,5428	8,2339	8,4364	8,0423	8,1763	2,1497	6,2516
3674,7	18,3609	8,9555	8,6529	8,8427	8,4745	8,5553	2,1607	6,4749
3858,3	19,3050	9,2446	9,0987	9,2821	8,9593	9,0578	2,3832	6,2932
4041,9	20,1835	9,9175	9,5810	9,6619	9,3279	9,5601	2,5349	7,1745
4225,6	21,1585	10,4443	9,9011	10,1498	9,7863	9,8869	2,6004	7,4431
4409,2	21,9956	10,7176	10,4015	10,7348	10,2432	10,4114	2,6513	7,6928
4592,8	22,9664	11,2342	10,7787	11,1166	10,6516	10,6747	2,7198	7,8357
4776,4	23,9049	11,7253	10,8951	11,5775	11,0890	11,1470	2,8339	8,1995
4960,1	24,6790	12,2745	11,3627	11,8273	11,5901	11,6481	2,9910	8,2544
5143,7	25,8649	12,5576	12,0864	11,9827	11,8616	12,1008	3,2207	8,8806
5327,3	26,6806	13,1141	12,5308	12,8522	12,3846	12,4355	3,3085	8,8519
5511,0	27,6906	13,6543	13,0435	13,1305	12,6987	12,9598	3,3170	9,2733
5694,6	28,4848	14,0253	13,4202	13,7287	13,2123	13,3807	3,5245	9,9925
5878,2	29,7061	14,4270	13,8410	14,1570	13,6163	13,6977	3,6697	9,5903
6061,9	30,3703	14,9153	14,1292	14,6229	14,1093	14,2673	3,5512	10,6757
6245,5	31,4519	15,3225	14,6427	14,8097	14,4089	14,6719	3,6706	10,2160
6429,1	32,3803	15,8539	14,9580	15,4344	14,8437	14,8034	3,7425	10,7778
6612,8	33,3609	16,0959	15,6368	15,9841	15,3886	15,5642	3,9880	11,2610
6796,4	33,9080	16,7807	15,8217	16,3347	15,8648	15,9582	4,1518	11,3680
6980,0	35,1054	17,2561	16,5481	16,6767	16,3000	16,4203	4,2800	11,3414
7163,7	35,8591	17,5912	16,4057	17,1577	16,5852	16,8457	4,4434	12,4470
7347,3	36,9188	18,1850	17,3202	17,7756	17,0721	17,2940	4,5414	12,4404
7530,9	37,7534	18,1754	17,7050	18,1021	17,5073	17,6065	4,5787	12,3769
7714,6	38,7284	18,9076	18,1481	18,6717	17,9295	18,0755	4,5228	13,3655
7898,2	39,7037	19,4031	18,6334	18,9275	18,2663	18,5168	4,6456	12,9091
8081,8	40,5025	19,8335	18,8586	19,2556	18,5141	18,8366	4,7914	13,8555
8265,5	41,6975	19,9896	19,3415	19,8995	19,1385	19,4667	5,1182	14,5524
8449,1	42,6193	20,6258	19,8722	20,4893	19,5821	19,8613	5,1819	14,2511
8632,7	43,1313	20,6505	20,3960	20,8694	20,0720	20,1155	5,1807	14,1298
8816,4	44,0392	21,5682	20,7409	21,0841	20,3184	20,7529	5,3866	15,1877
9000,0	45,1624	21,1768	21,2096	21,6141	21,0518	21,1474	5,3730	15,6864

B.2.2 RK4 procentuella körtider

Indata	version/interpreterad		version/referens				
Simulerad tid (simulearade sekunder)	Cython-kompilerat	Cython med C-funktionsanrop (REFERENS)	Returvärdes-deklarationer	Parameter-deklarationer	Retur- och parameter-deklarationer	Returvärdes-, parameter- och variabel-deklarationer	Variabel-deklarationer
2,0	48,22%	46,61%	101,68%	98,72%	99,87%	27,03%	74,06%
185,6	49,35%	47,12%	103,69%	99,09%	99,30%	25,89%	72,56%
369,3	48,92%	46,74%	101,53%	98,75%	100,06%	25,13%	71,95%
552,9	49,34%	44,95%	106,09%	102,70%	103,89%	27,17%	74,89%
736,5	48,33%	46,87%	102,18%	98,10%	97,78%	25,38%	70,26%
920,2	49,40%	47,26%	101,98%	99,72%	99,93%	26,33%	74,15%
1103,8	49,39%	46,98%	102,46%	98,81%	99,72%	25,10%	74,75%
1287,4	48,92%	46,83%	101,39%	99,69%	99,53%	25,13%	73,74%
1471,1	48,59%	46,21%	103,42%	100,99%	101,79%	26,80%	75,46%
1654,7	49,48%	47,17%	101,31%	98,66%	99,65%	26,27%	68,96%
1838,3	48,55%	46,48%	103,59%	99,69%	100,82%	26,20%	76,20%
2022,0	48,58%	46,69%	102,07%	98,74%	99,12%	24,77%	69,63%
2205,6	48,23%	47,09%	100,68%	97,48%	98,26%	26,57%	71,63%
2389,2	48,48%	46,03%	103,08%	99,33%	100,72%	27,07%	76,99%
2572,9	48,74%	47,33%	102,12%	98,47%	99,59%	25,96%	71,41%
2756,5	48,73%	46,76%	101,54%	98,72%	100,35%	25,12%	75,84%
2940,1	49,31%	46,68%	102,83%	99,26%	99,75%	25,60%	70,63%
3123,8	47,12%	46,59%	102,05%	99,16%	100,17%	25,62%	74,54%
3307,4	48,58%	46,76%	102,64%	98,52%	99,51%	26,24%	73,86%
3491,0	48,95%	47,18%	102,46%	97,67%	99,30%	26,11%	75,92%
3674,7	48,77%	47,13%	102,19%	97,94%	98,87%	24,97%	74,83%
3858,3	47,89%	47,13%	102,02%	98,47%	99,55%	26,19%	69,17%
4041,9	49,14%	47,47%	100,84%	97,36%	99,78%	26,46%	74,88%
4225,6	49,36%	46,79%	102,51%	98,84%	99,86%	26,26%	75,17%
4409,2	48,73%	47,29%	103,20%	98,48%	100,10%	25,49%	73,96%
4592,8	48,92%	46,93%	103,14%	98,82%	99,04%	25,23%	72,70%
4776,4	49,05%	45,58%	106,26%	101,78%	102,31%	26,01%	75,26%
4960,1	49,74%	46,04%	104,09%	102,00%	102,51%	26,32%	72,64%
5143,7	48,55%	46,73%	99,14%	98,14%	100,12%	26,65%	73,48%
5327,3	49,15%	46,97%	102,56%	98,83%	99,24%	26,40%	70,64%
5511,0	49,31%	47,10%	100,67%	97,36%	99,36%	25,43%	71,10%
5694,6	49,24%	47,11%	102,30%	98,45%	99,71%	26,26%	74,46%
5878,2	48,57%	46,59%	102,28%	98,38%	98,97%	26,51%	69,29%
6061,9	49,11%	46,52%	103,49%	99,86%	100,98%	25,13%	75,56%
6245,5	48,72%	46,56%	101,14%	98,40%	100,20%	25,07%	69,77%
6429,1	48,96%	46,19%	103,19%	99,24%	98,97%	25,02%	72,05%
6612,8	48,25%	46,87%	102,22%	98,41%	99,54%	25,50%	72,02%
6796,4	49,49%	46,66%	103,24%	100,27%	100,86%	26,24%	71,85%
6980,0	49,16%	47,14%	100,78%	98,50%	99,23%	25,86%	68,54%
7163,7	49,06%	45,75%	104,58%	101,09%	102,68%	27,08%	75,87%
7347,3	49,26%	46,91%	102,63%	98,57%	99,85%	26,22%	71,83%
7530,9	48,14%	46,90%	102,24%	98,88%	99,44%	25,86%	69,91%
7714,6	48,82%	46,86%	102,89%	98,80%	99,60%	24,92%	73,65%
7898,2	48,87%	46,93%	101,58%	98,03%	99,37%	24,93%	69,28%
8081,8	48,97%	46,56%	102,10%	98,17%	99,88%	25,41%	73,47%
8265,5	47,94%	46,39%	102,88%	98,95%	100,65%	26,46%	75,24%
8449,1	48,40%	46,63%	103,11%	98,54%	99,94%	26,08%	71,71%
8632,7	47,88%	47,29%	102,32%	98,41%	98,62%	25,40%	69,28%
8816,4	48,97%	47,10%	101,65%	97,96%	100,06%	25,97%	73,23%
9000,0	46,89%	46,96%	101,91%	99,26%	99,71%	25,33%	73,96%
Medelvärde:	48,77%	46,75%	102,44%	98,97%	99,96%	25,88%	72,84%

B.3.1 Dict filter mätdata

Indata	Körtid per version (s)					
Indata (antal nyckel-värdepar)	Interpreterad ren Python	Cython-kompilerat	Cython med C-funktionsanrop (REFRERENS)	Returvärdes-deklarationer	Parameter-deklarationer	Retur- och parameter-deklarationer
700	0.0022	0.0010	0.0010	0.0009	0.0009	0.0009
31297	0.1118	0.0613	0.0650	0.0596	0.0599	0.0590
61894	0.2387	0.1290	0.1310	0.1289	0.1292	0.1439
92491	0.4254	0.2521	0.2611	0.2683	0.2575	0.2524
123088	0.6326	0.4240	0.4074	0.3968	0.4011	0.3981
153685	0.8629	0.5693	0.5818	0.5658	0.6574	0.5647
184282	1.1317	0.7772	0.7885	0.7835	0.7794	0.7777
214879	1.4215	1.0150	1.0246	1.0021	1.0198	1.0160
245476	1.6442	1.1766	1.1777	1.1576	1.1678	1.1806
276073	1.9408	1.3721	1.3943	1.3688	1.3781	1.3663
306670	2.1764	1.5809	1.5895	1.5599	1.5777	1.5658
337267	2.3795	1.7680	1.7916	1.7610	1.8346	1.7745
367864	2.6955	2.0323	2.0463	2.0141	2.0308	2.0136
398461	2.9425	2.2227	2.2416	2.2071	2.2361	2.2150
429058	3.1932	2.4202	2.4475	2.3978	2.4131	2.3977
459655	3.4297	2.5997	2.6209	2.5871	2.8335	2.5883
490252	3.6791	2.7767	2.8195	2.7602	2.8271	2.7595
520849	3.8767	2.9411	2.9616	2.9396	2.9527	2.9247
551446	4.1537	3.1682	3.2084	3.1580	3.1804	3.1583
582043	4.4244	3.3470	3.3849	3.3480	3.3720	3.3519
612640	4.5889	3.4967	3.5748	3.4847	3.5050	3.4771
643237	4.8949	3.7342	3.7699	3.7678	3.7384	3.7080
673834	5.1090	3.9235	3.9598	3.9146	3.9947	3.9250
704431	5.4886	4.2530	4.2822	4.2158	4.2544	4.2227
735028	5.7240	4.4316	4.4699	4.4429	4.4464	4.4028
765625	5.9923	4.6163	4.7304	4.5911	4.6323	4.6169
796222	6.2678	4.8261	4.8568	4.7856	4.8183	4.7847
826819	6.4649	5.0110	5.0483	5.0116	4.9967	4.9740
857416	6.6913	5.1755	5.2349	5.1597	5.1985	5.1522
888013	6.9407	5.4302	5.5169	5.3442	5.4067	5.3502
918610	7.1786	5.5558	5.6365	5.5248	5.5784	5.5293
949207	7.4261	5.7770	5.8523	5.7507	5.8156	5.7417
979804	7.6889	6.0244	6.0397	5.9724	5.9761	5.9430
1010401	7.9679	6.1674	6.2285	6.1279	6.1682	6.1664
1040998	8.1963	6.4004	6.4101	6.3391	6.3566	6.3311
1071595	8.4186	6.5382	6.6112	6.5107	6.6000	6.5137
1102192	8.6891	6.7245	6.8206	6.6988	6.7492	6.7166
1132789	8.9274	6.9251	7.0605	6.9104	6.9607	6.9066
1163386	9.1809	7.1361	7.2230	7.0927	7.1494	7.0974
1193983	9.4377	7.3322	7.4176	7.3015	7.3433	7.3327
1224580	9.7082	7.5328	7.6330	7.4900	7.5369	7.4819
1255177	9.9512	7.7438	7.8109	7.6756	7.7296	7.6883
1285774	10.1460	7.9049	8.0724	7.9718	7.9302	7.8760
1316371	10.4232	8.1121	8.2640	8.0550	8.1272	8.0684
1346968	10.6376	8.3002	8.5026	8.2555	8.3226	8.2621
1377565	10.9179	8.5119	8.6659	8.4535	8.5169	8.4467
1408162	11.5199	9.0267	9.0783	8.9456	9.0000	8.9203
1438759	11.7202	9.1752	9.2840	9.1331	9.1808	9.1886
1469356	11.9423	9.3839	9.5672	9.3199	9.4084	9.3365
1499953	12.1508	9.5654	9.7504	9.5194	9.5888	9.5107

B.3.2 Dict filter procentuell körtid

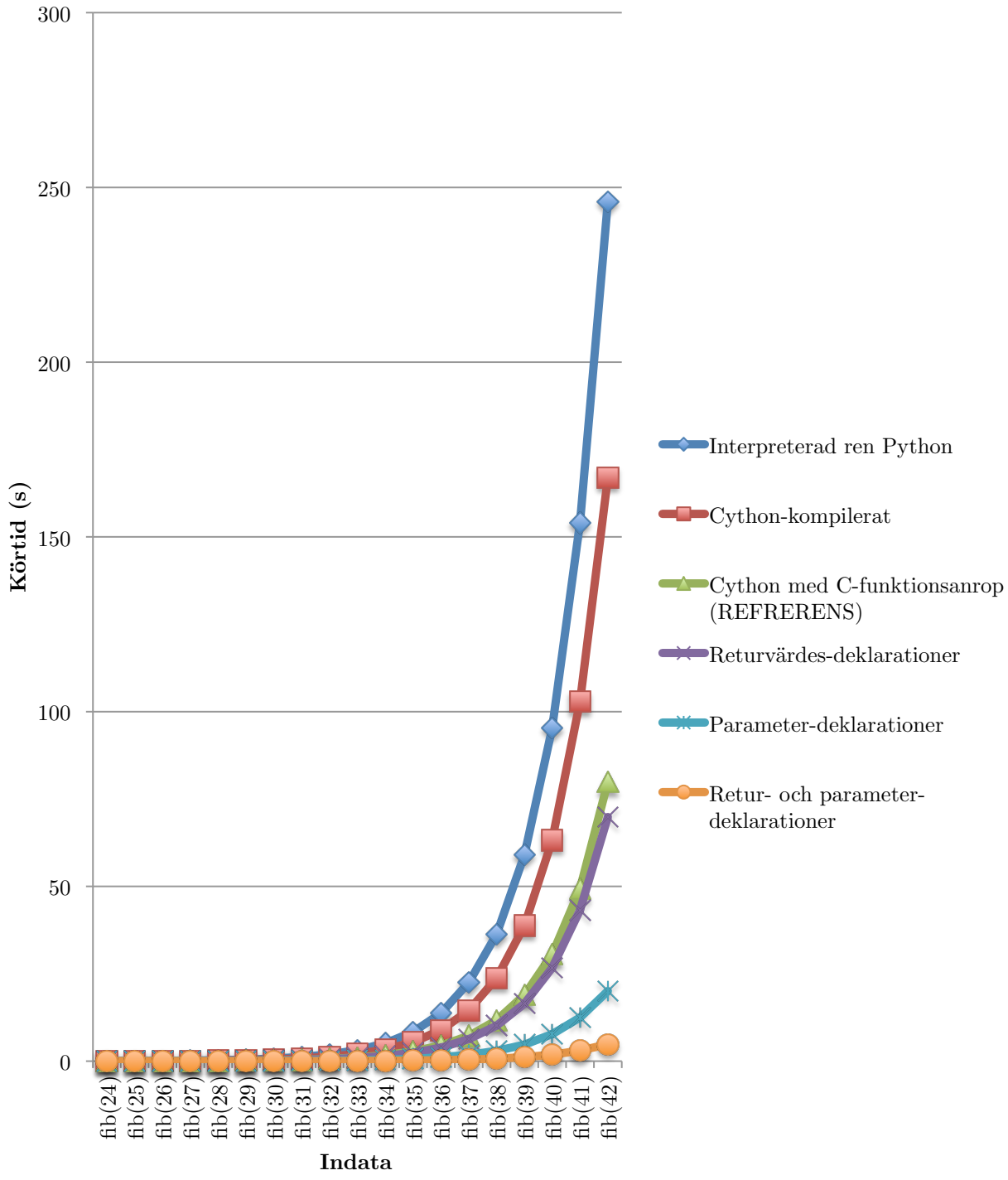
Indata	version/interpreterad		version/referens		
Indata (antal nyckel-värdepar)	Cython-kompilerat	Cython med C-funktionsanrop (REFERENS)	Returvärdes-deklarationer	Parameter-deklarationer	Retur- och parameter-deklarationer
700	45.03%	47.94%	90.65%	90.36%	89.59%
31297	54.86%	58.16%	91.62%	92.14%	90.77%
61894	54.05%	54.87%	98.42%	98.64%	109.87%
92491	59.26%	61.38%	102.75%	98.65%	96.68%
123088	67.04%	64.40%	97.40%	98.46%	97.73%
153685	65.98%	67.42%	97.24%	113.00%	97.06%
184282	68.68%	69.68%	99.36%	98.85%	98.63%
214879	71.40%	72.08%	97.80%	99.54%	99.16%
245476	71.56%	71.63%	98.29%	99.16%	100.24%
276073	70.70%	71.84%	98.17%	98.84%	97.99%
306670	72.64%	73.03%	98.14%	99.26%	98.51%
337267	74.30%	75.29%	98.29%	102.40%	99.05%
367864	75.40%	75.91%	98.43%	99.25%	98.40%
398461	75.54%	76.18%	98.46%	99.75%	98.81%
429058	75.79%	76.65%	97.97%	98.60%	97.96%
459655	75.80%	76.42%	98.71%	108.11%	98.76%
490252	75.47%	76.64%	97.90%	100.27%	97.87%
520849	75.87%	76.40%	99.26%	99.70%	98.75%
551446	76.28%	77.24%	98.43%	99.13%	98.44%
582043	75.65%	76.50%	98.91%	99.62%	99.03%
612640	76.20%	77.90%	97.48%	98.05%	97.27%
643237	76.29%	77.02%	99.94%	99.16%	98.36%
673834	76.80%	77.51%	98.86%	100.88%	99.12%
704431	77.49%	78.02%	98.45%	99.35%	98.61%
735028	77.42%	78.09%	99.40%	99.47%	98.50%
765625	77.04%	78.94%	97.05%	97.93%	97.60%
796222	77.00%	77.49%	98.53%	99.21%	98.52%
826819	77.51%	78.09%	99.27%	98.98%	98.53%
857416	77.35%	78.23%	98.56%	99.30%	98.42%
888013	78.24%	79.49%	96.87%	98.00%	96.98%
918610	77.39%	78.52%	98.02%	98.97%	98.10%
949207	77.79%	78.81%	98.26%	99.37%	98.11%
979804	78.35%	78.55%	98.89%	98.95%	98.40%
1010401	77.40%	78.17%	98.38%	99.03%	99.00%
1040998	78.09%	78.21%	98.89%	99.17%	98.77%
1071595	77.66%	78.53%	98.48%	99.83%	98.53%
1102192	77.39%	78.50%	98.21%	98.95%	98.47%
1132789	77.57%	79.09%	97.87%	98.59%	97.82%
1163386	77.73%	78.67%	98.20%	98.98%	98.26%
1193983	77.69%	78.60%	98.43%	99.00%	98.86%
1224580	77.59%	78.62%	98.13%	98.74%	98.02%
1255177	77.82%	78.49%	98.27%	98.96%	98.43%
1285774	77.91%	79.56%	98.75%	98.24%	97.57%
1316371	77.83%	79.28%	97.47%	98.35%	97.63%
1346968	78.03%	79.93%	97.09%	97.88%	97.17%
1377565	77.96%	79.37%	97.55%	98.28%	97.47%
1408162	78.36%	78.81%	98.54%	99.14%	98.26%
1438759	78.29%	79.21%	98.38%	98.89%	98.97%
1469356	78.58%	80.11%	97.42%	98.34%	97.59%
1499953	78.72%	80.24%	97.63%	98.34%	97.54%

Medelvärde:	74.22%	75.19%	98.07%	99.20%	98.16%

Bilaga C

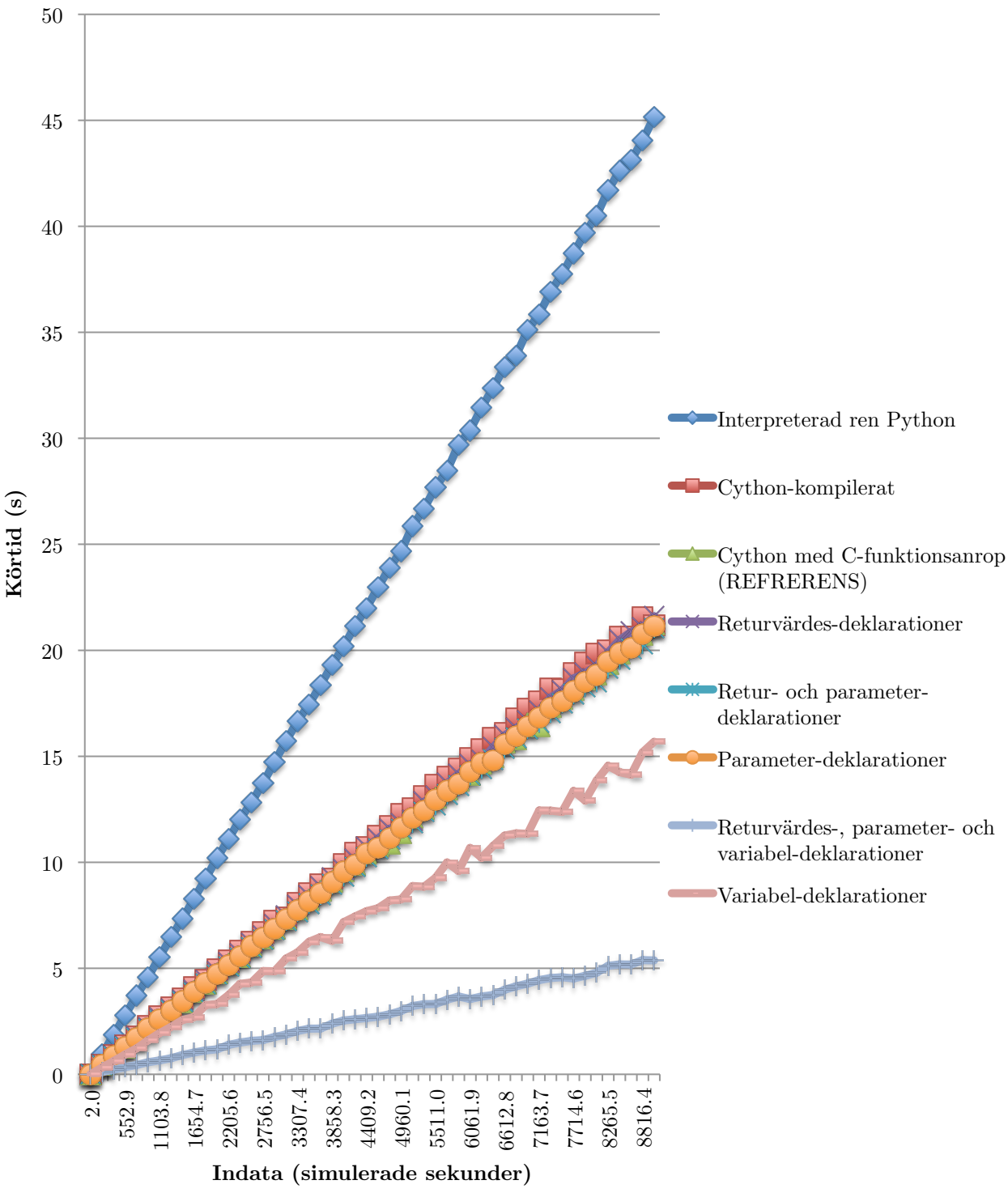
Diagram

C.1 Fibonacci körtider



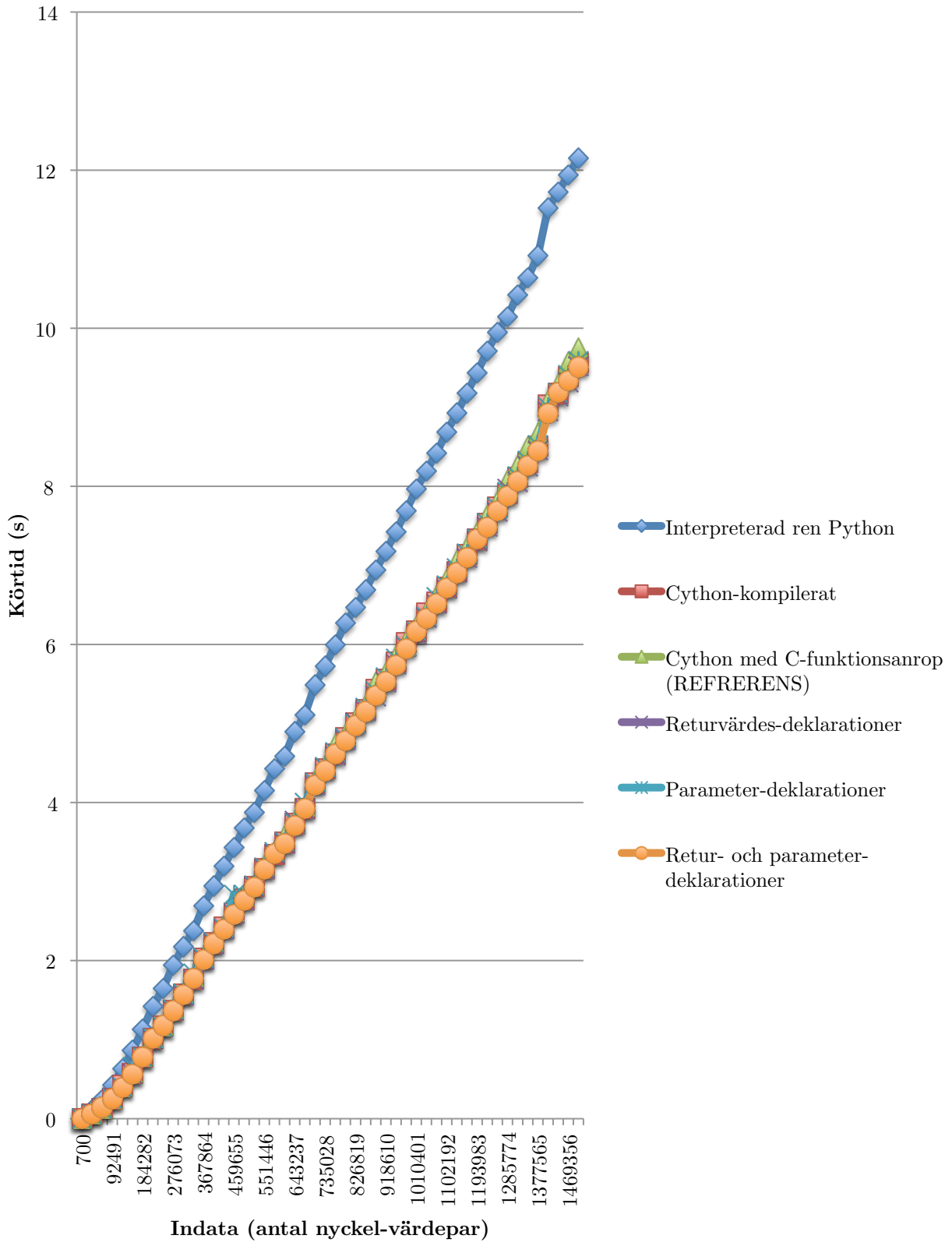
Diagrammet visar, för varje version, beräkningstiden för de olika fibonaccitalen.

C.2 RK4 körtider



Diagrammet visar, för de olika versionerna, körtiden för olika långt simulerade RK4-beräkningar.

C.3 Dict filter körtider



Diagrammet visar, för de olika versionerna, körtiden för filtrering av olika `dict` med ökande antal nyckel-värdepar.