Számítógépes modellezés „Julia” nyelvben

Mathematical modelling with the Julia language

KOPACZ Anikó, TASNÁDI Zoltán, dr CSATÓ Lehel

Babes-Bolyai Tudományegyetem,
Matematika és Informatika Kar,
RO-400084, M. Kogălniceanu u. 1 szám, Kolozsvár,
e-mail: {aniko.kopacz,zoltan.tasnadi,lehel.csato}@ubbcluj.ro, www: www.cs.ubbcluj.ro

Abstract

This article suggests that the newly developed language Julia – in the following called julialang – to be used as a tool for mathematical modelling. The language is based on other modern computer languages and is aimed to be at ease for mathematicians who can use it – together with its programming / coding environment – for implementing, understanding, and testing new algorithms. If necessary, the algorithms can be scaled up and run on a large – massive scale.

In the article we present the distinctive features of julialang – based on comparisons with the Python language and on some novel programming concepts -, followed by a few application domains that might make julialang a popular teaching tool. The article emphasises the ease of developing new algorithms in julialang.

Keywords: julialang, jupyter notebook, mathematical modelling, teaching modelling

Kivonat

Jelen írásban egy új nyelvet - a JULIA nyelvet - ismertetünk, melyet a továbbiakban julialang*-ként használunk. A julialang egy viszonylag "fiatal", de intenzíven fejlesztett nyelv, melyet feladatok matematikai modellezésére és azok gyors és skálázható tesztelésére optimalizálnak. A programnyelv kifejlesztésénél cél az is, hogy legyen elérhető egy kódoló-környezet - matematikusoknak eszköz -, melyet könnyen kezel egy matematikus; ebben a környezetben tud fejleszteni is új algoritmusokat; ezeket tudja tesztelni, illetve – amennyiben szükség – ezeket fel is tudja skálázni nagy rendszerekre.*

Bemutatjuk a nyelv jellemzőit - melyet a Python nyelv ismeretére, illetve a programozásból ismert funkcionális nyelvek és kompilátorok sajátosságaira alapozunk - majd bemutatunk néhány hasznos alkalmazási területet. Kiemelendő, hogy a julialang nagyon könnyen használható az oktatásban is, mely könnyedséget igyekszünk hangsúlyozni a bemutatás során.

Kulcsszavak: julialang programnyelv, jupyter, matematikai modellezés, számítógépes matematika, oktatás

1. BEVEZETŐ

A számítógépek és a számítástechnika gyors ütemű fejlődése mára már közhely; a számítási- és kommunikációs infrastruktúra a huszadik század közepétől rohamléptékben bővül. A fejlődés nagyvonalakban követi az 1960-as évek végén Moore által megfogalmazott skálázási törvényt[1] és a 2020-as években e „törvény” indukált változásainak a következménye, hogy – főképp külső szemlélő szerint – elképzelhetetlenül sok számítási és tárolási kapacitás áll rendelkezésre. A megnövekedett kapacitás; következésképp a megnövekedett adatok kihasználási szintje csökkenő; ez a csökkenő hatékonyság hasonlatos az 1970-es évek szoftverkrízisé-hez[2], amikor az akkori számítási kapacitás növekedésre válaszul megjelentek a programozási paradigmák:

[1] A Moore által megfogalmazott *exponenciális* fejlődés-görbe szerint a számítógépes kapacitás kétévente megkétszereződik. A link: https://www.britannica.com/technology/Moores-law (acc: 2022 okt. 1)

[2] A „szoftverkrízis” kifejezést 1968-ig lehet visszakövetni: a mérnökök nem tudták az elvárt minőségű és bonyolultságú szoftvereket legyártani az elvárt időben. A link: https://en.wikipedia.org/wiki/Software_crisis (acc: 2022 okt. 1)

előbb a strukturált programozás, majd az objektum-orientált programozás. Ezen paradigmák egyrészt sikeresek voltak a növekvő bonyolultságú rendszerek kifejlesztésében, másrészt lehetővé tették még nagyobb és komplexebb rendszerek létrehozását.

A növelt bonyolultságú rendszerek sokféle programozási és absztrakciós igényt kielégítettek: lehetett kiadványokat szerkeszteni[3], lehetett még bonyolultabb programokat tervezni[4], mint például a ma is közismert Visual Studio; ezekre, mint IDE hivatkozunk (*integrated development environment* – lásd 4. lábjegyzet). A továbbiakban a hangsúlyt a *matematikai – zömmel numerikus – fogalmak kezelésére* használt program-környezetekre helyezzük, ebből kiemelve két rendszert: a Mathematica és a Matlab környezeteket[5].

A fenti rendszerek jellemzői, hogy (1) biztosítanak egy „fejlesztői környezetet" az alapvető matematikai – numerikus – modellek implementálására *és* tesztelésére, illetve hogy (2) lehetővé teszik a megírt programkód – sok esetben a modellek egy-egy komponense – fejlesztés közbeni tesztelését; ez sok esetben részleges kiértékelést, mátrixok felületként történő megjelenítését, illetve más – intuitív vizualizációs megjelenítést feltételez; (3) ezek a rendszerek *platform-függetlenek*: futtathatóak különböző – elterjedt – operációs rendszereken[6].

A számítógépes rendszerek gyors fejlődésének egyik velejárója, hogy e rendszerek kiszolgáló hardvere nagyon heterogénné vált: nagyon sok számítógépes rendszerben másképp működnek az egyes komponensek, más típusú a rendelkezésre álló számítási kapacitás, illetve másképp lehet a számítási egységeket „ellátni" adatokkal annak érdekében, hogy a kapacitásukat ki tudjuk használni. Ez a heterogenitás azt jelenti, hogy – amennyiben a hagyományos, minden eszköz-kategóriára külön – kompilátort használunk, akkor a programozó feladata, hogy ezen különböző eseteket kezelje.

A programok fejlesztői munkáját megkönnyítendő, megjelentek az az „interpretált" nyelvek – ezt a „kompilált" nyelvek ellenpontjaként definiáljuk – melyek megkönnyítik a programozó munkáját és az adott számítási infrastruktúrán futtatják az általában nagyon magas szinten megírt programkódot. Ebbe az interpretált programozási nyelvek kategóriába tartozik a fent említett két programnyelv és IDE is: a Mathematica és a Matlab alapértelmezetten nem fordítanak gépi kódra, hanem az IDE-ben futtatható a megírt kód, ezáltal itt tesztelhető a modell. Ebbe a családba tartozik sok más – a Mathematica / Matlab nyelveknél jóval „fiatalabb" – programnyelv, a jelen cikk szempontjából számunkra a *Python* nyelv kiemelt[7]; egy interpretált nyelv, mely nagyon elterjedt, rendelkezik objektum-orientált alapokkal és napjaink „*deep learning*" alkalmazások alapja.

Az elkövetkezőkben egy – a python-nál is fiatalabb - új nyelvet mutatunk be, a *julialang*[8] nyelvet, melynek fejlesztése 2008-ban kezdődött. A fejlesztés egyik célja, hogy ez az új magas-szintű programozási nyelv képes legyen megosztott és párhuzamos programok leírására és futtatására, ugyanakkor a programkód közel legyen a matematikai nyelvezethez és ne tartalmazzon sok – matematikai szempontból – programozás-technikai sallangot, mellyel a kód futtatását ellenőrizzük. Az új nyelv célja volt tehát, hogy a nyelv érthető legyen az adatfeldolgozó algoritmusokat programozó matematikusok számára is.

A következőkben néhány olyan jellegzetességet sorolunk fel, melyek a klasszikus rendszereknél gondot jelentenek és a *julialang* esetében megoldás születhet erre.

1.1. A hardver sokszínűsége és a virtuális gép

A matematikai – modellezési, gépi tanulási, illetve „deep learning" típusú – rendszerek számításigénye nagy: egyszerűbb esetekben ezek nagy mátrixok összeszorzását jelentik, a mátrixok elemeire ugyanazon függvények alkalmazását, azonban bonyolultabb esetekben egy nagy *számítási gráf* ismételt végrehajtását kell elvégezze. Az ilyen számítási feladatokra adott válaszok sokfélék: egyrészt lehet egy központi számítási egységben – CPU-ban – több, logikai vagy fizikai mag, lehet egy alaplapon több CPU, lehet egy-egy alaplapot tartalmazó gépből sok, melyet egy *klaszter*-be csoportosítanak. Egy másik felosztás szerint pedig lehet a

[3] Ennek korai képviselői a DTP (desktop publishing) név alatt csoportosultak, jellemzői a WYSIWYG – az angol „what you see is what you get" rövidítése – és az 1980-as évek második felétől a Macintosh – későbbi Apple – számítógépek legfőbb alkalmazása (link: https://en.wikipedia.org/wiki/Desktop_publishing megtekintve. 2022. okt. 03)

[4] A programok tervezésének legfőbb kisegítője az „integrált fejlesztői környezet" – az IDE – integrated development environment (https://en.wikipedia.org/wiki/Integrated_development_environment megtekintve. 2022. okt. 03)

[5] Fontosnak tartjuk hangsúlyozni, hogy az általunk kiemelt Mathematica (link: https://www.wolfram.com/mathematica) és Matlab (https://www.mathworks.com/products/matlab.html) mellett sok rendszer biztosít valamekkora absztrakciót és környezetet, mellyel elősegíti a rendszerek tervezését és implementálását.

[6] Ezen operációs rendszerek, a Windows, a Linux, illetve az MacOs és az OSX.

[7] A *python* nyelv egy szintén interpretált nyelv, mely napjaink programozói eszköztárából kihagyhatatlan: a matematikai modellezés, a „deep learning" előtt és mellett a web-es rendszerek fejlesztésének is alapeszköze.

[8] Az angol *julia* programnyelvet alakítottam *julialang* formába, melynek a célja az egyértelműsítés: a rövid „go" megfelelője a *golang*, a „c" megfelelője a *Clang*; ily módon a megfeleltetés egyértelműbb.

műveleteket végezni CPU-n; de lehet azokat „kiszervezni” egy-egy GPU-ra[9], esetenként TPU-ra, de születnek újabb eszközök, melyek célja egy-egy speciális művelet-osztály optimalizált végrehajtása.

Egy válasz erre egy olyan virtuális gép megalkotása, amely a magaszszintű programkódot képes átalakítani arra a hardver-re, melyen akarjuk a rendszert futtatni. A kód futtatása így optimális lesz az adott rendszeren. Ez azt jelenti, hogy a kódba nem kerülnek hardverközeli utasítások, de ugyanakkor azt is jelenti, hogy a programkód más kritériumoknak is meg kell feleljen, melyek lehetővé teszik az utasítások – részleges – átrendezését az optimalizálás érdekében.

A *julialang* esetében a rendszer – az interpreter – egy futási-idejű kompilálást hajt végre, melynek célja nem a gépi kód, hanem egy virtuális gépen – melyet LLVM-nek hívunk – futtatható köztes kód – ezt az LLVM alakítja az adott hardveren futó bináris gépi kóddá.

Fontos megjegyezni, hogy ez a *virtualizáció* lehetővé teszi a *julialang* kódok optimalizálását és futtatását egészen kis – például egy Raspberry Pi – rendszertől kezdve egészen nagy rendszeren is. Ez utóbbira példa a „Celeste” projekt [lásd Regier et.al. 2016], melynek célja a megfigyelhető univerzum objektumainak osztályozása; az eszköz a Bayes-féle osztályozás –variációszámítási módszerekkel – volt és több, mint 9000 nagykapacitású számítógépen futott. Az LLVM és a *julialang* által képviselt programozási paradigmák egyik alkalmazása egy létező elektronikai szimulációs rendszer átírása úgy, hogy a rendszer *három nagyságrenddel gyorsabb* legyen[10]; a *julialang* az alkotói szerint ezzel a C-hez és a Fortran-hoz csatlakozott a nagyhatékonyságú számítógépes rendszerek programnyelvei közé.

Számunkra – itt kutatási és oktatási szempontokra hivatkozunk – fontos, hogy az LLVM-re történő kompilálás a *julialang*-ban történő fejlesztést teljesen platform-függetlenné teszi. Azt jelenti, hogy a kutatók számára nem szükséges a különböző architektúrákra való felkészülés; az oktatóknak és diákoknak pedig *csak* a fontos – algoritmikus szempontokra kell figyelniük – ezáltal növelve a kutatói és oktatói munka hatékonyságát.

1.2. A kétnyelv dilemma

A matematikai modellezés területén gyakori, hogy az algoritmusok megalkotásánál a programozói munka két részre tagolódik: egy első részen a modell „matematikai” részleteit rögzítjük és az algoritmust egy magasabb-szintű programozói környezetben fejlesztik, ekkor általában a Mathematica / Matlab a fejlesztés eszköze. Ebben a fázisban általában az algoritmus kifejlesztése zajlik; az adathalmazokat csökkentjük; a futási időt igyekszünk megbecsülni és a modell bonyolultságát becsülni. Amikor a modell és az ehhez kapcsolódó algoritmusok paramétereit meghatároztuk, akkor a fejlesztés folytatódik egy hatékony programnyelvben; erre általában a C/Fortran programnyelvek ideálisak. A hangsúly a programozási feladatok megkétszereződése az összes költséggel együtt.

```
r((x₁,x₂)) = (1-x₁)^2 + 100*(x₂-x₁^2)^2
∇r(x)   = gradient(r,x)[1]

x  = range(-2, 6, length=801)
ss = [r((x₁,x₂)) for x₂ ∈ x, x₁ ∈ x]
```

1. ábra: julialang programsorok: kiemeljük az UTF8 használatát és az automatikus lista-konstruktorokat (a 4. sorban).

A *julialang* esetében van remény arra, hogy ez a kettőzés megszüntethető: a megírt algoritmusokat – minimális változtatással – lehet használni a termelésben. Egy – apró, de nem teljesen elhanyagolható – szempont a programkódok formázása: a programozás során használhatjuk a teljes UTF8 karakterkészletet[11] – ahogy az 1. és 2. ábrákon láthatjuk – ez segíti a programok „matematikusabb” leírását; egy példa erre az „**r**” függvény két argumentuma, melyeket „**x₁**”, illetve „**x₂**”-ként definiálunk – jelképezve a matematikai eredetű *első és második koordinátát*. A második ábra tartalmazza a napjaink „deep learning” modelljeinek alapalgoritmusát – a gradiens-tanuló algoritmust, melynek paraméterei a tanuló adathalmaz, a „*feature*” függvény, a hibafüggvény deriváltja, illetve az algoritmus paraméterei.

```
function gradient_descent(𝒟tr, ϕ, ∇loss;
    η=0.1, T=100,
    w = zeros(length(ϕ(𝒟tr[1][1])))
    )
    for t in 1:T
        w -= η*mean(
            ∇loss(x,y,w,ϕ)
            for (x,y) ∈ 𝒟tr
        )
    end
    return w
end
```

2. ábra: a gradiens algoritmus a Beautifulalgorithms.jl csomagból.

1.3. A dolgozat szerkezete

[9] A „graphics processing unit” GPU – egy dedikált eszköz, mely egyszerű – matematikailag skaláris szorzatként definiált – műveleteket képes több nagyságrenddel gyorsabban elvégezni. Ehhez a sebességhez a GPU-nak szüksége van az adatoknak egy „kanonikus” elhelyezésére, illetve azok speciális – grafikus memóriának nevezett – helyre történő másolására.

[10] A link: https://juliacomputing.com/media/2021/08/DARPA, megtekintve: 2022 okt. 1.

[11] A kódszerkesztésnek egy módja a *jupyter notebook*; itt a LaTeX-szerű karakterláncot át tudunk alakítani UTF8 kóddá, például az 1. ábrán a „**∇**” jelet a „`\nabla`” leírásával majd a TAB karakterrel tudjuk elérni.

A cikk hátralevő részében bemutatjuk a *julialang* jellemzőit, kiemelve azokat, melyek miatt hatékony. A bemutatás folyamán kiemeljük azon jellemzőket, melyeket más programozási nyelvekből hasznosnak találtunk és a *julialang*-ban is jelen vannak. Bemutatjuk a *julialang* alkalmazását adatok feldolgozására és modellezésre. A cikk összegzésében felsoroljuk azon elemeket, melyek a *julialang* szempontjából előre mutatnak és megemlítjük azokat a jellemzőket, melyeken lehetne javítani. Megjegyezzük, hogy a vélemények nem objektívek és tükröznek egyfajta elfogultságot a „tiszta programkód”, illetve a „szép algoritmusok” iránt[12], amint azt a 2. ábrán láthatjuk.

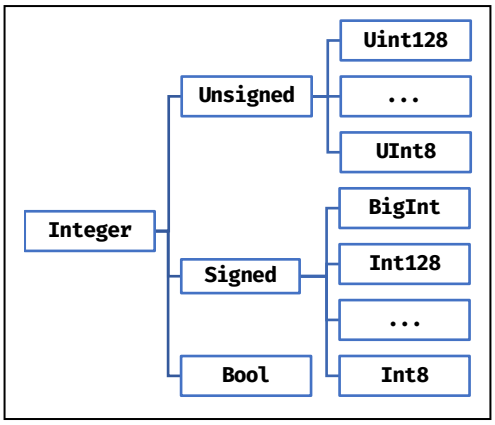
2. A JULIALANG-RÓL ÁLTALÁNOSAN

A *julialang* egy dinamikus programozási nyelv, melyben a változókat és függvényeket típusaik nélkül tudjuk kijelenteni – ebben a tekintetben a programnyelv hasonló a python-hoz, a Mathematica-hoz, illetve a Matlab nyelvhez. A fentiektől eltér azonban néhány tekintetben, az egyik a kijelentett változók *típusaira* vonatkozik: a *julialang*-ban van típuslevezetési mechanizmus, amely a fenti csoport programnyelveiből hiányzik, azonban ez a típuslevezetés nem annyira komplex – „okos” –, mint a funkcionális nyelvekben megszokott.

A nyelv sajátja egy típus-hierarchia (lásd a 3. ábrát), mely tartalmazhat parametrikus típusokat – ezek szintén a típusos modern funkcionális nyelvek sajátjai – és támogatja a struktúrák definícióját, mint új típusokat.

A típushierarchia a julialang esetében azt is jelenti, hogy a típusokról tudunk bennfoglalási relációkat kérdezni a típusokon értelmezett relációk segítségével. Meg tudjuk kérdezni például, hogy egy típus egy másiknál általánosabb vagy specifikusabb-e: `Number >: Integer` – igaz –; azt is, hogy adott típusoknak mik az altípusai – `subtypes` –, illetve azt, hogy mi az őse – `supertype`. Egy példát az `Integer` osztály altípusaival a 3. ábrán látunk.

A fentiek szerint tehát a julialang típusos, azonban *nem objektum-orientált* nyelv. A típus fogalma az adatok strukturálására vonatkozik és megengedett a *konstruktor* függvény, mellyel megadhatunk alapértelmezett értékeket egy-egy mezőnek, illetve meghívhatjuk a struktúra alkotóelemeinek a konstruktorait.



3. ábra: Julialang típus-hierarchia részlet: az `Integer` *altípusait soroltuk fel a* `subtypes` *rekurzív meghívásával.*

2.1. Változók, típusok, függvények és a *„multiple dispatch”* fogalma

A python nyelvhez hasonlóan, a julialang-ban a változókat ki lehet jelenteni egy megfeleltetéssel és akkor, amikor a változót használni fogjuk, azaz amikor *szükségünk* van erre a változóra. Értelemszerűen – szintén a *python*-hoz hasonlóan – amikor a változónak értéket adunk, akkor ennek *típusa* is lesz, melyet az értelmező vezet le. A levezetés eredménye egy *konkrét – nem absztrakt* – típus, jelölése – a funkcionális nyelvekhez hasonlóan[13] – a „`valt::Típus`” jelöléssel adjuk meg. A típusok nagybetűvel kezdődnek, ez javasolt a saját típusokra is.

```
f1(x) = x+1
# lambda-kifejezés
f2 = x -> x+1
#függvénytest definícióval
function f3(x)
    x+1
end
```

4. ábra: Példa függvények kijelentésére a julialang nyelvben: „matematikailag”, funkcionálisa, illetve lambda-jelöléssel.

A függvényeket a mellékelt – 4. ábrán illusztrált – módozatok egyikén: amennyiben a függvény „egyszerű”, akkor elégséges megadni a kimenő érték kifejezését – ez a 4. ábra első sorában található. Amennyiben nem csak egy kifejezés a függvény, akkor használhatjuk a `function` és az `end` kulcsszavakat a függvény megírására, illetve használhatjuk a „lambda-jelölést” amennyiben egy magasabbrendű függvény argumentumaként szeretnénk megadni azt. Vegyük észre, hogy egyik függvény sem tartalmazza az argumentumainak a típusát.

```
class Num a where
   (+), (-), (*) :: a -> a -> a
   negate        :: a -> a
   abs           :: a -> a
   signum        :: a -> a
```

5. ábra: Példa haskell típusra: a `Num` *típus esetében felsoroljuk a* ***típus által indukált*** *– használható – függvényeket.*

A *julialang* érdekessége, hogy – hacsak nem specifikáljuk explicit módon – a formális változók típusa az `Any`; azaz a függvények teljesen generikusak – a *python* nyelv-beli definíciókhoz hasonlóan –, ez pedig

[12] Egy gondosan megírt algoritmus-csomag a *Beautifulalgorithms.jl*, egy részletet a 2. ábra tartalmaz.
[13] A haskell és a Clean nyelvekben a „`valt::Típus`” kijelentéssel definiáljuk adott változó típusát.

ellentétes a *modern funkcionális nyelvekben* elfogadott típuslevezetéssel. A *haskell* esetében például a típuslevezetések illeszkednek a „legspecifikusabb” típusosztályra, melyek segítségével az adott függvény összes művelete értelmezett (lásd a 6. ábrát); az adott függvény végrehajtható. A *julialang* ezzel szemben a típuslevezetésnél *„lusta”*, ez a „negatív lustaság”: a rendszer egyszerűen nem végez típusellenőrzést, a julialang esetében a típushierarchia tetején levő – a legáltalánosabb – `Any` absztrakt típust felelteti meg.

A típuslevezetés a *haskell* nyelv esetében a – a mellékelt 5. ábrán látható módon – a függvényeken keresztül valósítható meg: a *típusosztályt* akkor adjuk a típushoz megszorításként, amikor a típusosztály egy függvényét alkalmazzuk az adott változóra.

Mivel a *julialang* esetében az absztrakt típusokhoz nem társulnak függvények, a pontosabb típusmegszorítások automatikus levezetése sem megvalósítható. Itt ugyanis a típusokat csak a felhasználók látják el funkcionalitással; a julialang esetében függvények nevei mellett nyilvántartja a rendszer a függvény argumentumainak a típusát. Híváskor azt a függvényt hívja meg a rendszer – hasonlóan, de nem ugyanúgy, mint az objektum-orientált programozás esetén az öröklődés és a virtuális metódusok mechanizmusához – amelyik típusa a *legközelebb áll* az aktuális paraméterhez. Ezt a mechanizmust nevezzük „*multiple dispatch*”-nek.

A „*multiple dispatch*” illusztrálására tekintsük az 6. ábrán mellékelt kódot , ahol definiáljuk a `z` függvényt típusmegszorítások nélkül, majd ugyanezzel a névvel definiáljuk a `String`-et duplázó `z` függvényt. Ekkor a julialang `methods` függvénye – szintén az 6. ábrán illusztrált módon – felsorolja a „`z`” függvényeit a paramétereihez rendelt típusokkal (az alapértelmezett `Any` nincs feltűntetve a felsorolásban).

A „*multiple dispatch*” tehát a *julialang* függvényhívásának a mechanizmusa, mely által a futási időben állapítjuk meg azt, hogy a `methods`-ban felsorolt függvények közül melyik lesz végrehajtva [lásd még a **3.** hivatkozást]. Érdekes kísérlet például a hatványozás műveletének implementálása; ugyanis látványosan másképp kell elvégezni az egész, illetve a valós számok hatványozását: amennyiben „közelítő” értéket szeretnénk, akkor használhatjuk a `(3.0)^45` kifejezést, ennek közelítő értéke `2.95431e21`, amennyiben pontos értéket szeretnénk, akkor a `(BigInt:3)^45`, azaz `2954312706550833698643`, ez „természetesen” hibás lesz az alapértelmezett 64 bites tárolás esetén: `2833654757305440083`.

```
# első kijelentés
z(x) = x + 1
# típusosan
z(x::String) = x^2
# ekkor a függvények
> z(x::String) in Main at ...
> z(x) in Main at ...
```

6. ábra: Típusok által vezérelt „multiple dispatch”: a z függvénynek azt a példányát fogjuk meghívni, amelyik típusa közelebb van az aktuális paraméterhez.

A változókról szóló rész végén megemlítjük, hogy a julialang sem használ memória foglalást / felszabadítást, mutatókat, hanem a memóriakezelést a „*garbage collection*”[14] módszerrel oldja meg: a változók kijelentésénél történik a memória-foglalás; amikor a rendszer úgy dönt, hogy szüksége lenne korábban lefoglalt tárhelyre, felszabadítja azon lokációkat, melyekre már nincs hivatkozás.

A „*garbage collection*” és a dinamikus memória-kezelés az ismert *python* nyelvhez teszi hasonlóvá a *julialang* nyelvet. A dinamikus változó-kijelentés könnyíti ugyan a programozást az adott nyelven, ez a „haladó programozóknak” ugyanakkor gondot jelent, ugyanis a szigorúan típusos nyelvek esetében a típusinformáció sokat segít a kódok megértésében és a programok helyességének a megállapításában.

A többi interpretált nyelvhez hasonló módon, a *julialang* esetében is lényeges hátrányt jelent az, hogy a programozói hibákat csak futási időben látjuk. Továbbá gond, hogy nem tudunk felszabadítani memóriát, hanem a rendszerre kell bízni; ezért a dinamikus memóriakezelők a programok futási sebességének a szórását nagyon megnövelik.

2.2. Összetett típusok, struktúrák definíciója és a konstruktorok

A julialang esetén is tudunk – a C/C++ -hoz hasonlóan – új típusokat létrehozni, melyet a 7. ábrán illusztrált módon tehetünk meg. A julialang jellemzője a strukturáltság és az, hogy *nem objektum-orientált*.

A julialang típusrendszere dinamikus – futási időben is megállapítható egy változó típusa – és hasonlít a modern funkcionális nyelvek típusaira. Az egyszerű típusok mellett jelen vannak a származtatott típusok – mint például a `Tuple` – ami típusokat csoportosít[15] –, amennyiben névvel szeretnénk ellátni, akkor a `struct` kulcsszavat használjuk, a 7. ábrán illusztrált módon.

[14] A *garbage-collection* – „szemétgyűjtés” – a dinamikus memória-kezelés egyik sarokköve, ahol a rendszer dönt arról, hogy adott változónak mikor foglal új memóriát, azt mikor szabadítja fel, és hogyan kezeli a bonyolultabb struktúrákat és referenciákat.

[15] Típusokra lekérdezésére a `typeof((1,2.5))` függvényt használjuk, az eredmény `Tuple{Int64,Float64}`.

Annak ellenére, hogy nem objektum-orientált a nyelv, az OOP-ből ismerős konstruktor ellenben jelen van; ezzel tudjuk specifikálni a struktúrák létrehozásának a részleteit. A julialang-ban a struktúrák mezői „csomagolva" vannak: amikor létrehozunk **`struct`**-tal változót, a mezők *immutábilisak*: a **`p::Point{Int8}`** létrehozása után a **`p.x`** és a **`p.y`** változók nem változtathatóak. Ez hasznos az „egységbezárás" illusztrálására és a konstruktorok által ellenőrzött megszorítások betartására (lásd alább), és amennyiben az immutabilitást nem szeretnénk, a struktúra definíciójánál kell megtennünk a „**`mutable`**" kulcsszó beszúrásával.

```
struct Point{T <: Real}
    x :: T
    y:: T
    Point{T}(x,y) where {T<:Real} = new(x,y)
    Point{Int32}(x,y) = new(1,1)
end
```

7. ábra: Új – paraméteres – típusok létrehozása: a **`Point`** *egy „`T`" típusú koordinátapárt jelent; megszorítás, hogy a koordináták típusai azonosak kell, hogy legyenek.*

A nyelvi konstrukciót ki lett terjesztve arra az esetre is, amikor a struktúra elemeire megkötéseket teszünk: definiálhatunk rendezett párosokat – a 7. ábra **`Point`** struktúrájához hasonlóan – ahol az első attribútum *mindig* kisebb mint a második; ekkor a konstruktor csak akkor jut el az allokációig, ha a kért feltétel teljesül: **`Pair(x,y) = x>y ? error("out of order") : new(x,y)`**.

A **`struct`** kulcsszó struktúrákat definiál a julialang nyelvben, azonban a nyelv „specialitása" a listákkal, vektorokkal és a mátrixokkal történő műveletek. Ennek alaptípusa az **`Array`**, melyet használunk listák kezelésére és számsorok tárolására egyaránt.

2.3. Vektorok, mátrixok és műveletek mátrixokkal

Az **`Array`** a julialang alaptípusa, elemek felsorolását és elemeként történő feldolgozást tesz lehetővé. Az **`Array`** egy *származtatott típus* – a mátrix (vektor, vagy tenzor) elemeinek a típusát jelentjük ki. A 8. ábra két listakijelentést tartalmaz: az **`x_s`** egy *generátor*[16], míg az **`ss`** változó egy mátrix – kétdimenziós tömb – amit az ábrán látható listagenerátor[17] hozott létre.[18] A létrehozott **`Array`** típusú változó minden eleme egy számpár, az **`ss`** változó pedig kétdimenziós; ezt a **`for`** kulcsszó utáni két iterátor mutatja. A jelölés matematikai formalizmust tükröz: az „$x_1 \in$ **`xs`**" az „x_1" változónak rendre megfelelteti az **`x_s`** mindegyik elemét; a listagenerátorban specifikált függvényt elvégzi minden esetben, a kimeneteket pedig elmenti a **`Matrix`** struktúrában.[19]

```
# listagenerátor
x_s = range(-2,6, length=101)
# párosok mátrixa
ss  = [ (x₁,x₂)
    for x₂ ∈ x_s,
        x₁ ∈ x_s
    ]
```

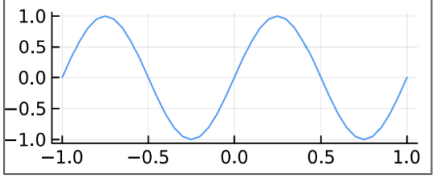
8. ábra: Listák létrehozása generátorokkal: az **`x_s`** *változóban a generátort tároljuk, az* **`ss`** *változóban a struktúrát, melyet a lista-generátor segítségével hoztunk létre.*

Egy érdekes – a julialang kiemelt jellemzője – a függvények vagy műveletek „*broadcast*"-elése egy **`Array`** – lehet vektor vagy nagyobb dimenziós entitás – típusú változó összes elemére. Ez matematikailag egy struktúra minden elemére alkalmazott függvény, a jelölése pedig a függvény neve utáni – vagy a műveletjel előtti – pont. Példa erre a 9. ábrán található kód: az ábrán az **`x`** változó egy generátor, melyet instanciálunk a **`sin`** értékek számításánál. Vegyük észre, hogy a jelölés „kifejthető" az alábbi kód-sorba:

`y = [sin(2π*z) for z ∈ x]`

Amint látható, a 9. ábra második sora kifejezőbb és rövidebb.

```
x = -1:0.05:1  # x - lokáció
y = sin.(2π*x) # a "broadcast"
# kirajzolás
plot(x,y,label=false,size=(300,120))
```



9. ábra: Példa a „broadcast" művelet alkalmazására: a „`sin`*" függvényt 41 „*`x`*" értékre kell kiszámolni; a* **`plot`** *ábrázolja.*

A továbbiakban egy példán keresztül ábrázoljuk a julialang korábbi jellemzőinek – a „*multiplex dispatch*" és a „*broadcast*" mechanizmusnak a következményeit. Tekintsük a négyzetes – 2x2-es – mátrixokon értelmezett hatványozás műveletét és az exponenciális függvényt, legyen a mátrixunk: $A = \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix}$; az A mátrixra a hatványozás más eredményt ad abban az esetben, ha elemenként alkalmazzuk és más eredményt ad abban az esetben, ha a hatványra emelést, *mint mátrixművelet*et használjuk. A mátrixműveletként értelmezett hatványozás eredménye:

[16] Az **`x_s`** típusa: **`StepRangeLen{Float64, TwicePrecision{Float64}, TwicePrecision{Float64}, Int64}`**.

[17] A 8. ábrán látható *listagenerátor* angol neve a „list comprehension", mely tükrözi a nevében, hogy a konstrukció a listák létrehozásának a megkönnyítését szolgálja – segítve egyszerre a műveletek könnyebb megértését is.

[18] Az **`x_s`** típusa **`Matrix{Tuple{Float64, Float64}}, (alias for Array{Tuple{Float64, Float64}, 2})`**.

[19] Fontos megjegyezni, hogy az iterátorok argumentumainak a specifikálására tudjuk használni az „**`in`**" kulcsszót is, azonban a szimbólumok használata hozzásegít a kód gyorsabb megértéséhez.

$B = A^2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$,
míg a mátrix négy elemére egyenként alkalmazott négyzetre emelés a
$C = A\,.\hat{}\,2 = \begin{bmatrix} 0 & 9 \\ 0 & 0 \end{bmatrix}$
mátrixot eredményezi; a két eset teljesen különböző eredményt ad. A julialang esetében a két változat közötti különbség jelölése a pont, amellyel specifikáljuk a „*broadcast*” műveletet, mely közel van a matematikai jelöléshez:

```
B = A^2, C = A .^ 2
```

ahol a „**.**” az atomi műveletvégzés jele (hasonlóan a Matlab-beli jelöléshez). Hasonlóan működnek a más függvények is, melyeknek létezik skaláris-, illetve mátrix-változata is. A példa az exponenciális függvény, mely mást jelent a mátrix elemeire egyenként alkalmazva; és mást a mátrix egészére. A fenti példán:
$D = \exp(A) = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$, illetve $E = \exp.(A) = \begin{bmatrix} 1 & 20.09 \\ 1 & 1 \end{bmatrix}$.

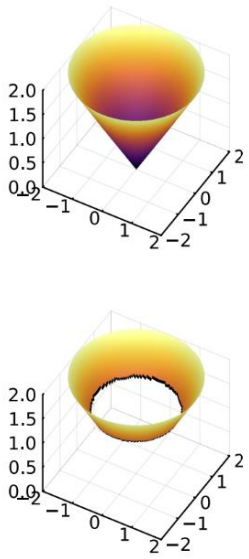
Fontos megjegyezni, hogy a műveleteket és a függvényeket *túlterheltük*: a julialang „*multiple dispatch*” mechanizmusa felelteti meg a különböző eseteknek.

Egy fontos jellemzője ezen műveleteknek a „logikai indexelés” egyszerűsített mechanizmusa, ahogyan a mellékelt 10. ábrán láthatjuk: a „*broadcast értékadás*” – az **`s[s .< 1.7] .= NaN`** művelet – előbb létrehoz egy logikai mátrixot és *csak* azon elemekhez rendeli hozzá a **`NaN`** értéket, melyekre a logikai feltétel igaz. A 10. ábrán előbb kiszámítjuk egy pont normáját, megjelenítjük, majd „töröljük” azokat, melyek normája kisebb adott értéknél. A megjelenítés látványos: az alsó ábrán a fenti teljes kúpnak csak a fenti része látszik.

```
# az euklidészi norma
norm((x₁,x₂)) = sqrt(x₁^2 + x₂^2)

ss2 = norm.(ss);
p = plot(layout=(2,1))
surface!(p[1], x_s,x_s, ss2)

# broadcast értékadás
ss2[ss2 .< 1.3] .= NaN
surface!(p[2], x_s, x_s, ss2)
```



10. ábra: a **`norm`** *függvény „broadcast”-elt formája:* **`norm.(ss)`***; a logikai indexelés broadcast-elése:* **`s[s .< 1] .= NaN`**.

Összefoglalva tehát, a julialang úgy lett megtervezve, hogy a matematikai fogalmakat a matematikusok számára minél érthetőbben tárolja. Ennek nyelvi eszköze a kiterjesztett szimbólumkészlet használata – láttuk, hogy a gradienst, a görög betűket, az kisbetűket tudtuk jelölni „natívan”. A funkcionális nyelvekből „örökölt” lambda-függvény jelölés egy újabb absztrakciós szintet tesz lehetővé: a **`filter`**, **`map`**, **`sum`** függvények ezen lambda-függvényekkel kombinálva nagyon lecsökkentik a programsorokat. További fontos eszköz a listagenerátorok használata és az automatikus „*broadcast*” mechanizmus.

2.4. Modulok a függvények logikai csoportosítására

A modulokat a függvények logikai csoportosítására használjuk, a python és más „modern” nyelvekhez hasonlóan. A julialang specifikuma, hogy a modulokat platformfüggetlenné teszi – a lehetőségek szerint.[20] A különböző funkcionalitásokat más-más modul importálásával tudjuk használni. A „külső” funkcionalitásokat két módon tudjuk importálni a rendszerünkbe: az **`import`** kulcsszóval az illető modul összes függvényét elérhetővé tesszük, azonban a függvények hívásánál ki kell írjuk a modul nevét. Példa a nagyon felhasználóbarát módon megírt **`Plots`** modul, ahol az **`import Plots`** használatával a kirajzolásokat a **`Plots.plot`** szintaxissal kell használnunk. A másik lehetőség a „**`using Plots`**” modul-importálás; ebben az esetben a modulok nevét nem kell kiírni; elégséges a – például 10. ábrán látható – **`surface`** függvény meghívása. Jelen dolgozatban a hangsúlyt a modulok használatára helyezzük – a modulok megírását és definícióját az olvasó a szakirodalomban találja, például a [2,3] referenciákban. A csomagokat kezelni a julialang keretében a *„pkg”* módban lehet.[21]

A rendszerben előbb a *regiszterek* listáját frissítjük – letöltjük az elérhető csomagokat – az „**`update`**” paranccsal, majd az „**`add`**” paranccsal tudunk új modulokat hozzáadni [22]

[20] A julialang-ot futtatni a felhasználói módban lehet – nem kell admin/root jog – és ez érvényes a Windows, a Linux, és az Apple és az ARM rendszerekre (ez utóbbiaknál az alacsonyszintű – és olcsó – adatfeldolgozást tesz lehetővé).

[21] A *konzol*-ban a „**`;`**”-vel egy „*shell módot*” indítunk, a „**`]`**” karakterrel a modulok kezelését tudjuk végezni, a „**`?`**” karakterrel a „*help”* üzemmódot kezdeményezünk.

[22] Amennyiben egy modul nincs – még – a rendszerben, a julialang rendszer felajánlja az installálást, tehát a létező funkcionalitások és csomagok használata egyszerűsödik. Alternatívaként installálásra a **`Pkg`** csomagot is használhatjuk.

3. FONTOSABB JULIALANG CSOMAGOK

A következőkben felsorolunk néhány csomagot, melyek hasznosak a programozás során. Megemlítjük, hogy a julialang alapcsomagja a *Base*, ez tartalmazza az általánosan használt függvényeket. A csomagok felsorolásban a személyes preferenciákat tükrözik.

LinearAlgebra – tartalmazza a fontosabb többdimenziós mátrix-művelet implementációját. Ilyenek például a mátrixok determinánsának a kiszámítása, az inverzének, a sajátérték-vektoroknak, a különböző dekompozíciós módszereknek a kódja – az LU decompozíció, a Cholesky-dekompozíció. Nagyon sok esetben a numerikus módszerek ismert LAPACK [5] függvénygyűjteményét implementálja. Definiálja a Matlab-ból ismert „\" operátort, mely az $Ax = b$ lineáris egyenletet oldja meg az `x = A \ b` szintaxissal. Az előnye, hogy az „`A`" mátrixot nem kell invertálni; a művelet négyzetes csupán.[23]

Random, Distributions – a véletlenszám-generátor csomag. Felülírja a *Base* csomag `rand` függvényét. A csomagban lehetőség van mintavételezésre diszkrét és folytonos eloszlások szerint; lehetőség van egy mintahalmaz alapján megállapítani egy eloszlás optimális paramétereit – a *fit* függvény segítségével.

DataFrames, CSV – az adatok beolvasására és gyors megtekintésére írt csomagok. Egy fontos adattárolási típus, a CSV-file-ok beolvasási mechanizmusa: `DataFrame(CSV.read(fileName))`, ahol a `DataFrame` a csomagban definiált típus. Táblázatok feldolgozásához tartozó műveleteket definiál; hasonlóan a python nyelv *pandas*[24] csomagjához. Amennyiben numerikusak az adataink, a `Matrix` – a C/C++ nyelvhez hasonló – típuskonverzióval mátrixszá alakítjuk, majd numerikus módszerekkel feldolgozzuk.

IJulia, Conda – a *jupyter* notebook futtatásának csomagja. Az „interaktív julia" – szintén a python-hoz hasonlóan – egy web-szervert indít, majd a böngészőben tudunk indítani *notebook*-okat. A notebook-ok cellákra osztjuk, ezeket egyenként tudjuk futtatni és a adatelemzői munka fontos eszköze – iteratívan tudunk algoritmust fejleszteni és tesztelni. Az oktatási folyamat fontos eszköze mivel a *notebook*-ban tudjuk vegyíteni a julialang kódot az adatok vizualizációjával, valamint az algoritmusokat leíró képletekkel és magyarázatokkal.

Plots, Plotly, GR, PyPlot – a julialang adatvizualizációs csomagjai. A *Plots* egy generikus adatvizualizációs csomag, melyben lehetőségünk van a különböző – operációs rendszer-, illetve dokumentum-típus függvényében – formában megjeleníteni: a PDF-től a JPG ábrán keresztül a web-es formátumoknál használt interaktív javascript formában, a különböző megjelenítési formák közötti váltáshoz „csak" a *Plots* egy-egy függvényének a meghívása szükséges.[25]

Flux – a gépi tanulás és a „deep learning" algoritmusainak a gyűjteménye. Lehetőségünk van – tanulási adatok alapján – egy függvény paramétereinek az optimalizálására, amennyiben pedig szeretnénk használni a grafikus magokat, a CUDA csomaggal tudjuk ezt megtenni anélkül, hogy a kódot lényegesen kellene változtatnunk.[26]

4. MEGOLDOTT FELADAT

A továbbiakban illusztráljuk a feladatok megoldását julialang-ban. A bemutatóhoz a julialang által is biztosított jupyter notebook interfészt használjuk – itt a cellák definiálják a logikailag összetartozó parancssorokat, azokat egyenként futtatjuk; a létrehozott változók a futtatás után rendelkezésünkre állnak; használhatjuk azt vizualizációkhoz. illetve más változók előállítására.

A példa-feladat során generálunk *zajos adatokat* és egy *modellt* illesztünk az adatokhoz. Az adatokat egy függvény – a kódban $f(x) = 2x + 1$ – értékei adják: az „`x`"-ekhez tartozó kimenő értékhez hozzáadjuk a megfigyelési zajt – ennek normál eloszlása és szórása $\sigma_n = 0.2$, ezekből pedig gyűjtünk „`n_tr`" számút.

```
using Plots, Random, LinearAlgebra
using Distributions, DataFrames, CSV

# adatgenerálás paraméterei
n_tr, n_te = 9, 6
fn = x -> 2x+1 ;
σₙ = .2

# adatok generálása
x_tr = rand(Uniform(0,1),n_tr)
x_te = rand(Uniform(0,1),n_te)
y_tr = fn.(x_tr) .+ σₙ * randn(n_tr)
y_te = fn.(x_te) .+ σₙ * randn(n_te)
```

11. ábra: Tanuló adatok generálása: a csomagok kijelentése, a paraméterek definíciója, majd az adatgeneráló kód.

[23] A *dynamic dispatch* mechanizmussal a különböző mátrixok – diagonális, sáv-diagonális, Toeplitz-mátrixok – esetén a lineáris rendszerek megoldását optimalizáljuk.

[24] A *pandas* csomag a „python nyelv adatelemző modulja", mint azt a csomag honlapján olvashatjuk https://pandas.pydata.org, megtekintve: 2022 okt. 1.

[25] A *Plots* csomagnak kitűnő dokumentációs oldala van: https://docs.juliaplots.org, illetve az adatvizualizáció más csomagjait is el tudjuk érni a Plots-on keresztül – például a `boxplot`-ot a *StatsPlots*-ból, megtekintve: 2022 okt. 1.

[26] A *Flux* dokumentációs oldala a https://fluxml.ai, példaprogramokat és kapcsolódó csomagok dokumentációja felé is tartalmaz, megtekintve: 2022 okt. 1.

A kódsorokat a 11. ábrán láthatjuk. megemlítjük, hogy a rendszer egyik alapfüggvénye a **`rand`** függvény, mely alapértelmezetten egy **`[0,1]`** közötti számot térít vissza, azonban a Random csomagban ezt felüldefiniáljuk és lehetőség van különböző eloszlások szerinti mintavételezésre – a 11. ábrán illusztráltuk ezt a **`Distributions`** csomagban definiált **`Uniform(a,b)`** eloszlással.

Az adatgenerálást követően kijelentjük az adatábrázoló függvényt – a 12. ábrán –, mely megjeleníti a generált *tanuló-* és *teszt-adathalmazt*. Az adatok kis száma lehetővé teszi, hogy meg tudjuk figyelni az egyes algoritmusok jellemzőit. Amint látjuk a 12. ábra grafikonján, az adatok a generátor függvény körül vannak, azonban a zaj miatt nem pontosan a függvényen.

A közelítéshez másodfokú modelleket használunk. A 13. ábrán definiáljuk a „*feature*" -függvényt, mely generálja – mindegyik „**x**" komponensre – a polinomiális bemeneti értékeket; a közelítés felírható ugyanis a $pred(x,\theta) = \Phi(x) \cdot \theta$ alakban, ahogy ezt a **`pred_fn`** függvény definíciójánál is látjuk. A korábban említett julialang jellegzetességet kiemeljük: a kompilátor tudja – a „**Φ**", illetve a **`pred_fn`** függvények definíciójánál a típusdefinícióból – hogy a függvényeket *csak* adott típusú paraméterrel futtatja a rendszer – a paraméterünk egy vektor, melynek elemei a **`Real`** absztrakt típusnak a leszármazottjai.

A 14. ábrán kiválasztunk a korábban generált 9 pont közül ötöt, majd ezekre illesszük a másodfokú görbéket. Ennek a műveletnek a kódja a „\" – melyről beszéltünk több alkalommal; legutóbb a **`LinearAlgebra`** csomag felülírt operátorainál – és amely a legkisebb négyzetes hibával történő paraméter-becslést valósítja meg. A becsléseknek az eredménye a 14. ábra lenti részén látható.

Összefoglalva, a julialang egy kitűnő programozási nyelv, mely dinamikusan típusos nyelv, ugyanakkor kitűnően használható modellezési feladatok megoldására és az oktatásban is.

KÖSZÖNETNYILVÁNÍTÁS

A szerzők köszönik az „Erasmus+ stratégiai partnerség program" támogatását, melynek neve „*Promoting Sustainability as a Fundamental Driver in Software Development Training and Education* (Sustrainable)"; kódszáma: 2020-1-PT01-KA203-078646.

IRODALMI HIVATKOZÁSOK

[1] Regier, J., Pamnany, K., Giordano, R., Thomas, R., Schlegel, D., & McAuliffe, J. (2016). *Learning an astronomical catalog of the visible universe through scalable Bayesian inference*. arXiv preprint arXiv:1611.03404.

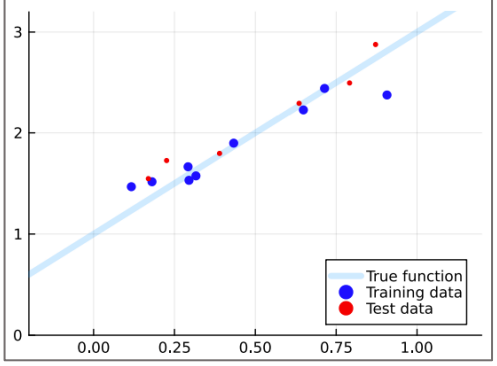
[2] Sherrington, M. (2015). *Mastering Julia*. Packt Publishing Ltd.

[3] Balbaert I. (2015) *Getting started with Julia programming*. Pack Publishing, ISBN 978-1-78328-479-5.

[4] Sengupta A. (2019) *Julia high performance. Optimisations, distributed computing, multithreading, and GPU programming*, Pack Publishing, ISBN 978-1-78829-811-7

[5] Anderson E, Bai Z, Bischof S C andBlackford, Demmel J, Dongarra J J andDu Croz, Greenbaum A, et al.(1999) *LAPACK Users' Guide*, Third Edition.Society for Industrial and Applied Mathematics, ISBN: 0-89871-447-8.

```julia
# a megjelenítő függvény
function plot_data(fn, (x_tr,y_tr), (x_te,y_te);
    p = plot()
)
    plot!(p, fn,
        alpha=0.2,width=5,xlim=(-.2,1.2),
        label="True function"
    )
    scatter!(p, x_tr,y_tr,
        color=:blue,ms=4,msw=0,
        label="Training data"
    )
    scatter!(p, x_te,y_te,
        color=:red,ms=2.3,msw=0,
        xlim=(-.2,1.2),ylim=(0,3.2),
        label="Test data",legend=:bottomright
        ,size=(400,300)
    )
end
plot_data(fn, (x_tr,y_tr), (x_te,y_te))
```



12. ábra: Tanuló adatok generálása: a csomagok kijelentése, a paraméterek definíciója, majd a generáló kód.

```julia
# másodrendű jellemzők
Φ(x::Vector{T}) where T<:Real = hcat(
    ones(length(x)), x, x.^2
)
pred_fn(
    x::Vector{T},
    θ::Vector{T}
) where T <: Real = (Φ(x) * θ)[1]
```

13. ábra: A modell és az előrejelző függvény definíciója.

```julia
p = plot()
n_data = 5
n_exp  = 300
for k ∈ 1:n_exp
    i_exp = randperm(n_tr)[1:n_data]
    x,y = x_tr[i_exp], y_tr[i_exp]
    θ = Φ(x) \ y
    plot!(p, x -> pred_fn([x],θ), xlim=(-.3, 1.3),
        label=false,width=.5,alpha=.3
    )
end
plot_data(fn,(x_tr,y_tr), (x_te,y_te),p=p)
p
```

14. ábra: 300-szor kiválasztunk 5 mintát, tanítjuk a rendszert – a „\" operátorral (fent), és megjelenítjük a függvényt (lent).