MASARYKOVA UNIVERZITA

PŘÍRODOVĚDECKÁ FAKULTA

DIPLOMOVÁ PRÁCE

Ondřej Kutal

Tvorba matematické grafiky pomocí programu Asymptote

Vedoucí práce: RNDr. Roman Plch, Ph.D.

Studijní program: Matematika

Studijní obor: Matematika s informatikou

BRNO 2012

Masarykova univerzita

Přírodovědecká fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Ondřej Kutal**

Studijní program - obor: **Matematika - Matematika s informatikou**

Ředitel Ústavu matematiky a statistiky PřF MU Vám ve smyslu Studijního a zkušebního řádu MU určuje diplomovou práci s tématem:

Tvorba matematické grafiky pomocí programu Asymptote

Mathematical graphics with the program Asymptote

Oficiální zadání: Popište instalaci a použití programu Asymptote při tvorbě matematické grafiky. Zaměřte se zejména na spolupráci s LaTeXem a na tvorbu PDF dokumentů s vloženou interaktivní 3D grafikou, popište matematický základ algoritmů pro generování této grafiky. Vygenerujte interaktivní 3D grafiku pro podporu výuky Integrálního počtu funkcí více proměnných.

Literatura:

Plch, Roman - Šarmanová, Petra. Interaktivní 3D grafika v HTML a PDF dokumentech. Zpravodaj Československého sdružení uživatelů TEXu, Praha : Československé sdružení uživatelů TEXu, 18, 1-2, od s. 76-92, 16 s. ISSN 1211-6661. 2008.

Plch, Roman - Šarmanová, Petra - Sojka, Petr. Integrální počet funkcí více proměnných. Elportál: portál Masarykovy univerzity [online], Brno : Masarykova univerzita, 2009, 1, 160 s. ISSN 1802-128X. 2009.

Vedoucí diplomové práce: RNDr. Roman Plch, Ph.D.

Datum zadání diplomové práce: listopad 2009

Datum odevzdání diplomové práce: dle harmonogramu ak. roku 2010/2011

V Brně dne 10. 11. 2009

v. z.

prof. RNDr. Jiří Rosický, DrSc.
ředitel Ústavu matematiky a statistiky

Zadání diplomové práce převzal dne: 2. 12. 2009

Podpis studenta

Poděkování

Rád bych touto cestou poděkoval RNDr. Romanu Plchovi, Ph.D., za odborné vedení diplomové práce, za cenné rady a připomínky a za čas, který mi věnoval při konzultacích.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů.

V Brně dne 20. února 2012 Ondřej Kutal

Název práce: Tvorba matematické grafiky pomocí programu Asymptote
Autor: Ondřej Kutal
Ústav matematiky a statistiky Přírodovědecké fakulty, MU
Vedoucí diplomové práce: RNDr. Roman Plch, Ph.D.
Abstrakt: Cílem této práce je popis tvorby matematické grafiky s programem Asymptote s důrazem na 3D interaktivní grafiku v dokumentech PDF. Snaha textu je, aby z pestrých příkladů čtenář pochytil základní myšlenky práce s programem a zároveň měl dostatek informací o pozadí těchto postupů. V první části jsou vysvětleny základní syntaktická pravidla, v dalších částech je pak popsána metodika práce ve 2D a 3D. Na konci práce jsou rozebrány algoritmy, které v Asymptote převádějí rovinné útvary do 3D reprezentace. V příloze jsou pak uvedeny příklady grafických objektů pro podporu výuky Integrálního počtu funkcí více proměnných.
Klíčová slova: Asymptote, 3D grafika, PDF, LaTeX

Title: Mathematical graphics with the program Asymptote
Author: Ondřej Kutal
Department of Mathematics and Statistics, Faculty of Science, MU
Supervisor: RNDr. Roman Plch, Ph.D.
Abstract: The aim of this diploma thesis is to describe creating of mathematical graphics with the program Asymptote with emphasis on 3D interactive graphics in PDF documents. The task of this text is to make learning of the program easier with various examples used, while at the same time providing a solid description of the background of techniques used. In the first part, the basic syntax rules are explained, then in following chapters the routines for 2D and 3D are described. In the last chapter, algorithms converting planar regions into 3D surfaces are explained. Then an examples to support the teaching of Integral Calculus of Multivariable Functions are provided in appendix.
Keywords: Asymptote, 3D graphics, PDF, LaTeX

Obsah

Úvod

Cílem této práce je představit a popsat program Asymptote, účinný a pohodlný open source nástroj pro tvorbu 2D i 3D (interaktivní) grafiky, vhodný především pro vědecké dokumenty. Asymptote je interpretovaný jazyk se syntaxí založenou na jazyce C++. Jeho potenciál je vidět především při psaní dokumentů v jazyce LaTeX, při kterém můžeme umisťovat bloky zdrojových kódů obrázků do speciálních prostředí, a grafiku tak generovat společně se samotným dokumentem. Od roku 2004 na něm pracují pánové John C. Bowman, Andy Hammerlindl a Tom Prince. Od té doby se Asymptote stále vyvíjí, každým rokem přibývají nové funkcionality a opravy chyb. Původně byl program koncipován jako alternativa k systému METAPOST, oproti kterému dovoluje například přesnější výpočty v plovoucí desetinné čárce či tvorbu interaktivní grafiky ve 3D.

V první kapitole je popsána instalace a nastavení programu Asymptote, včetně základních způsobů práce s programem. Ve druhé kapitole jsou popsány základy syntaxe programu. V další části jsou pak vysvětleny postupy pro generování 2D a 3D grafiky, včetně mnoha příkladů. Jelikož jsou však možnosti této práce omezené, doporučuji se podívat na další příklady do [1, Gallery] či [3]. V poslední kapitole je popsán algoritmus pro vytváření 3D reprezentace rovinných oblastí na základě jejich popisu křivkami. To se v Asymptote využívá kdykoliv používáme LaTeXové popisky ve 3D, ale samozřejmě nejen tam. V příloze jsou pak vypracovány některé 2D obrázky a 3D modely z [5]. Pro všechny ukázky v práci byla použita verze Asymptote 2.14.

Kapitola 1

Instalace

Před instalací Asymptote je potřeba mít:

- TEX implementaci
  např. TeXLive, lze stáhnout na `http://www.tug.org/texlive/`
- Ghostscript GPL
  viz `http://sourceforge.net/projects/ghostscript`
- nějaký prohlížeč formátů `.ps` a `.pdf`
  např. GSview, Adobe Reader (pro správné zobrazení 3D scén v `pdf` je potřeba Adobe Reader 8.0 a vyšší)

Pro systémy Windows pak z oficiálních stránek Asymptote (viz [1]) musíme stáhnout samotný Asymptote, například v podobě samorozbalovacího souboru `asymptote-x.xx-setup.exe`, kde `x.xx` značí aktuální verzi programu. Pro uživatele linuxových systému jsou k dispozici `tgz` i `rpm` balíky, pro podrobnější informace k vaší distribuci viz [1, Documentation-2.1 UNIX binary distributions]. Po instalaci je často potřeba provést nastavení souboru `config.asy`, viz část 1.1.

1.1 Nastavení `config.asy`

Po nainstalování Asymptote je vhodné provést nastavení, a to především cesty k jednotlivým prohlížečům, TEXové implementaci, případně jiné parametry. Nastavení se provádí v souboru `config.asy`, který se nachází ve složce `.asy` v uživatelově domovské složce (na systémech Windows se nachází v `%userprofile%`), případně se dá nastavit alternativní cesta využitím systémové proměnné `ASYMPTOTE_HOME`.

Většinu systémových cest Asymptote při instalaci nastaví sám, na linuxových systémech tato starost většinou úplně odpadá. Někdy je přesto potřeba něco nastavit ručně. Důležité jsou především cesty k PostSriptovému prohlížeči (psviewer), GhostScriptu (gs) a prohlížeči PDF (pdfviewer). Na linuxových systémech vše zpravidla funguje po instalaci, na systémech Windows by nastavení config.asy mohlo vypadat například takto:

```
import settings;

gs="C:\Program Files\Ghostscript\gs9.00\bin\gswin32c.exe";
psviewer="C:\Program Files\Ghostgum\gsview\gsview32.exe";
pdfviewer="C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.
    exe";
```

Pro více informací o možnostech nastavení souboru config.asy viz [1, Documentation-2.4 Configuring].

1.2 Parametry příkazové řádky

Asymptote je možné spustit s různými parametry, které budou mít planost pouze pro dané spuštění. Většina z těchto parametrů se dá nastavit i v souboru config.asy, a to použitím delšího názvu parametru (pokud má i zkrácenou verzi). Tabulka 1.1 shrnuje seznam některých parametrů (úplný seznam se zobrazí při spuštění Asymptote s parametrem -h).

-a,-align C|B|T|Z	zarovnání na stránce, Center, Bottom, Top, nebo Zero [C]
-autoplay	automatické spouštění 3D animací [false]
-embed	vloží renderovaný náhledový obrázek pro 3D scénu [true]
-h,-help	zobrazí seznam všech parametrů; pouze pro příkazovou řádku
-f,-outformat format	nastavení výstupního formátu
-o,-outname name	nastavení cesty pro výstupní soubor
-prc	vloží do pdf dokumentu 3D PRC model [true]
-tex engine	latex|pdflatex|xelatex|tex|pdftex |context|none [latex]
-toolbar	ukáže 3D toolbar v PDF výstupu [true]
-version	zobrazí verzi Asymptote
-l,-listvariables	vypíše seznam funkcí a proměnných [false]
-V,-View	zobrazí výstupní obrázek; pouze pro příkazovou řádku

Tabulka 1.1: Parametry příkazové řádky Asymptote

Pro vypnutí parametru stačí před název vložit `-no`, tj. například

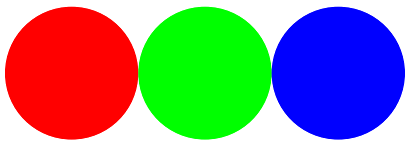
```
asy -f pdf -notoolbar
```

nastaví výstupní formát `pdf` a ve výsledném dokumentu skryje pruh nástrojů pro 3D scénu.

1.3 Interaktivní mód

Do interaktivního módu Asymptote se dostaneme spuštěním `asy` v příkazové řádce systému. Pak stačí zadávat příkazy a potvrzovat klávesou Enter. Výstupní grafický soubor se pak ukládá do souboru `out.eps` v aktuálním adresáři (pokud nespecifikujeme jiný výstupní formát). Pokud máme správně nastavený PostScriptový prohlížeč (proměnná `psviewer` v souboru `config.asy`), uvidíme výstupní soubor automaticky při každé jeho změně (případně po kliknutí na obrázek nebo stisknutí klávesy R ve většině prohlížečů).

```
> size(0,50);
> fill(circle((-2,0),1),red);
> fill(circle((0,0),1),green);
> fill(circle((2,0),1),blue);
```



Obr. 1.1: Ukázka výstupu

Zadávání dalších příkazů přidává objekty k dosud vykreslenému. Pro vyčištění kreslící plochy slouží příkaz `erase`.

Užitečná je klávesa Tabulátor, při jejímž stisknutí Asymptote doplní název funkce či proměnné (pokud je to jednoznačné), nebo nabídne možná doplnění. Zde je tabulka základních příkazů pro interaktivní mód:

`help`	zobrazí nápovědu
`erase`	vyprázdní plátno
`reset`	vrátí nastavení do původního stavu
`quit/exit`	ukončí program

Tabulka 1.2: Příkazy pro interaktivní mód

1.4 Dávkový (batch) mód

Dávkový mód se provede spuštěním `asy` s parametrem názvu souboru, ve kterém máme připravený kód. Asymptote za nás potom přečte sekvenci příkazů a vygeneruje grafiku do výstupního souboru. Implicitně je výstup ve formátu `eps`, ale vhodnou volbou parametrů (viz část 1.2) lze nastavit výstup do jiných formátů[1]. Pokud například máme připravený kód v souboru `krivka.asy`, můžeme vygenerovat grafický výstup takto:

```
asy krivka.asy -V
```

kde parametr `-V` zajistí zobrazení výstupu v prohlížeči po ukončení.

1.5 Použití v LaTeXu

Pro použití Asymptote uvnitř LaTeXového dokumentu musíme mít správně umístěny příslušné LaTeXové balíky. K tomu, pokud to za nás neudělala instalace, musíme zkopírovat soubory `asymptote.sty`, `asycolors.sty` a `ocg.sty` do místa, kde LaTeX hledá balíky. V dokumentu pak načteme balík `asymptote` příkazem:

```
\usepackage[inline]{asymptote}
```

Nyní můžeme použít prostředí `asy` a do něj umístit zdrojový kód obrázku.

```
\begin{asy}
...
\end{asy}
```

Další možností je mít kód v samostatném souboru a vkládat ho pomocí LaTeXového příkazu `\asyinclude` (například `\asyinclude{elipsa.asy}`).

Pro kompilaci grafiky pro dokument použijeme posloupnost příkazů

```
pdflatex -interaction=nonstopmode dokument.tex
asy -batchView dokument-*.asy
pdflatex -interaction=nonstopmode dokument.tex
```

(případně místo `latex` můžeme použít `cslatex` či `pdfcslatex`). Používáme-li nějaké prostředí pro psaní LaTeXových dokumentů, máme v nich někdy k dispozici tlačítko pro Asymptote (nebo dokonce pro posloupnost všech tří příkazů), jindy si jej musíme vytvořit (a nebo kompilovat obrázky přes příkazovou řádku). Prostou obměnou předchozích příkazů můžeme generovat

---

[1] Asymptote podporuje výstupní formáty `eps`, `pdf`, ale také `jpg`, `png` či `svg`, je však potřeba nainstalovat program ImageMagick pro konverzi, případně `dvisvgm`, více viz [1, Documentation-9 Command-line options]

dokument do formátu **ps** (ovšem bez 3D grafiky), jen bychom místo předchozího napsali:

```
cslatex -interaction=nonstopmode dokument.tex
dvips -o dokument.ps dokument.dvi
asy -batchView dokument-*.asy
cslatex -interaction=nonstopmode dokument.tex
dvips -o dokument.ps dokument.dvi
```

Pro dokumenty s větším obsahem grafiky se může hodit utilita `latexmk`, pomocí níž se kompilují jen změněné obrázky (viz [1, Index-latexmk]). Případně je možné „ručně“ zkompilovat jen změněné obrázky běžným způsobem pro samostatné `asy` soubory. Pokud například víme, že pozměněná grafika se nachází pouze v souboru `dokument-2.asy`, použijeme

```
asy dokument-2.asy
```

a tím překompilujeme jen zvolený obrázek. Při příštím použití `pdflatex` (či podobného programu) se nová grafika vloží do dokumentu.

Při kompilaci dokumentu s vloženou grafikou je generováno velké množství dočasných souborů. Ty je někdy nutné promazat, a proto se hodí oddělit dočasné soubory od těch důležitých. To si můžeme usnadnit použitím LaTeXové proměnné `asydir`, která definuje podsložku pro dočasné soubory. Pro takové použití stačí vytvořit složku `temp` a na začátek dokumentu přidat:

```
\renewcommand\asydir{temp}
```

a pak pouze mazat obsah složky `temp`. V takovém případě se však musí danému LaTeXovému prostředí říct, aby spouštělo Asymptote v složce `temp`, protože v původní složce nic nenajde. Toto nastavení záleží na konkrétním prostředí.

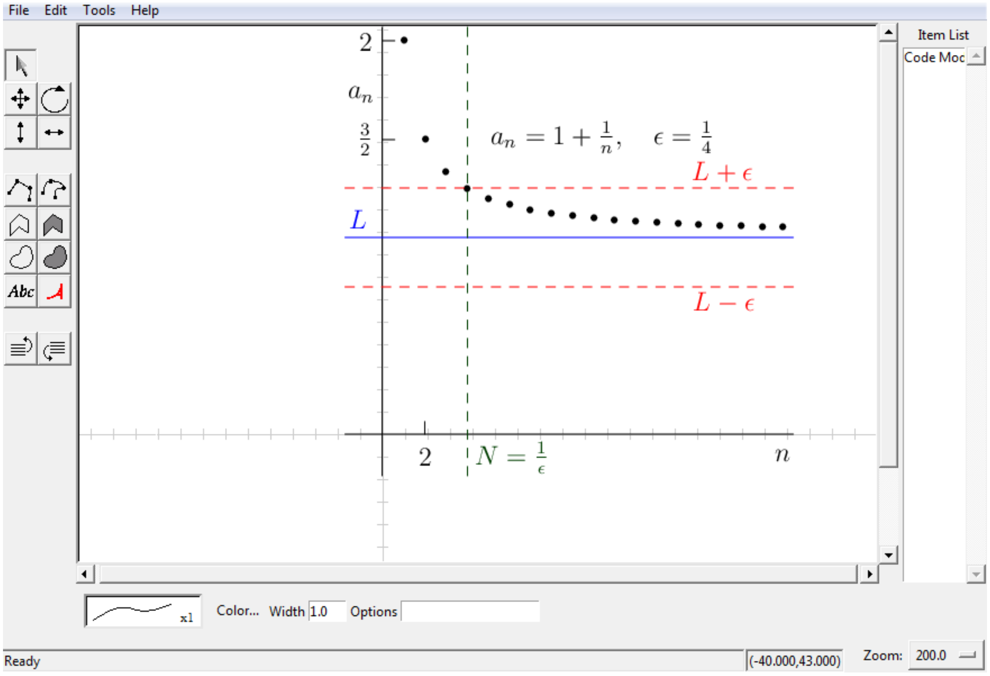
Jiný způsob, jak oddělit důležité soubory od těch dočasných, je umístit je do podsložky např. `files`, a v původní složce mít pouze hlavní TeXový dokument a dočasné soubory. Pak stačí mazat všechny soubory kromě hlavního dokumentu a složky `files`.

LaTeXový balík `asymptote.sty` také definuje prostředí `asydef`, do kterého můžeme vkládat globální nastavení pro celý dokument. Jedná se o obyčejné příkazy v Asymptote, tj. je možné mít nastavené například:

```
\begin{asydef}
settings.prc = false; //bez interaktivni 3D grafiky
\end{asydef}
```

1.6 Xasy

Asymptote v sobě obsahuje jednoduché uživatelské rozhraní Xasy, které je napsané ve skriptovacím jazyce Python, jehož interpret je potřebný pro běh (`http://www.python.org`). Samotné Xasy zatím nabízí pouze omezené možnosti kreslení, hodí se spíše pro dolaďování již existujících obrázků.



Obr. 1.2: Grafické rozhraní Xasy

1.7 Problémy

Práce v Asymptote vždy neprobíhá tak hladce, jak by si člověk přál. Proto si zde uveďme nejčastější problémy a způsoby jejich řešení. Zaměřme se nejdříve na situaci, kdy nám ani nevznikne výstupní grafický soubor.

Při práci v LaTeXu při tomto problému doporučuji nejdřív promazat všechny dočasné soubory vytvořené Asymptote. Nejedná se však pouze o soubory `.asy`, ale také některé `.tex`, `.ps`, `.pdf` a jiné soubory. To si můžeme usnadnit například použitím LaTeXové proměnné `asydir`, viz část 1.5.

Pokud nezabralo smazání dočasných souborů, jedná se zřejmě o syntaktickou chybu či zapomenutí nějakého balíku. V takovém případě je užitečné generovat grafiku z příkazové řádky a mít celý kód ve zvláštním `.asy` souboru. Pak jednoduše uvidíme chybová hlášení a měli bychom být schopni chybu odhalit. Případně můžeme zvýšit úroveň výpisů pomocí parametrů -v,-vv až -vvvvv. Asymptote dovoluje i ladění z příkazové řádky, včetně nastavení breakpointů apod. Více lze nalézt v [1, Documentation-Debugger].

Dalším častým zdrojem problémů bývá chybné či nedostatečné nastavení souboru config.asy. Může se jednat o nesprávnou cestu k souborům TEXového kompilátoru (MikTeX, TeXLive), Ghostscriptu či samotných souborů Asymptote, více viz část 1.1. Na závěr si uveďme řešení některých dalších problémů.

Kompilace se zastaví při zpracovávání programem Ghostscript, na výstupu se objeví seznam čísel a program čeká na potvrzení klávesou Enter uživatelem.
Tento problém se stává na systémech Windows při generování 3D grafiky s TEXovými popisky. Problém je způsobený použitím gswin32.exe místo gswin32c.exe v konfiguračním souboru config.asy. Po změně by měl problém zmizet.

Vše se zkompiluje, ale při prohlížení v Adobe Readeru je scéna po aktivaci příliš přiblížená.
Zde se jedná o chybu Javascriptu v linuxových verzích Adobe Readeru a projevuje se v systémech s neanglickým prostředím. Stačí spouštět Adobe Reader pomocí příkazu:

```
LC_NUMERIC=C acroread
```

Pro širší nastavení lze na vhodné místo do souboru /usr/bin/acroread vložit řádek:

```
export LC_NUMERIC="C"
```

V případě, že se ani uvedenými způsoby nepodaří problém vyřešit, je stále velká šance, že někdo měl již podobný problém. Doporučuji proto diskusní fórum Asymptote (viz [1, Forum]) a často pokládané otázky (viz [1, FAQ]).

Kapitola 2

Syntaxe

Asymptote používá syntaxi velice podobnou jazykům C/C++ a Java, jsou zde však odlišnosti. Tuto sekci probereme podrobněji, aby i čtenář neznalý jazyků C, C++ či Java mohl Asymptote využít. Přitom však předpokládáme znalosti alespoň základních principů programování.

2.1 Základy

Soubor se zdrojovým kódem Asymptote se skládá z posloupnosti příkazů oddělených středníkem. Příkazem

```
import balik;
```

lze načítat balíky pro nejrůznější potřeby. Balíky nejsou ve skutečnosti nic jiného, než zdrojové soubory Asymptote, většinou dodávané tvůrci programu. Předchozí příkaz by tak vložil zdrojový kód ze souboru `balik.asy`. Takové soubory obsahují definice pokročilých typů a funkcí. Podrobnější informace o balících lze nalézt v části 2.7.

Vkládání komentářů se provádí stejně jako v C/C++. První možností je použít značku `//`, která zahajuje komentář a následný text do konce řádku je tak při kompilaci ignorován. Pro rozsáhlejší komentáře slouží dvojice značek `/*` a `*/`, označujících postupně začátek a konec komentáře.

```
/*
Zde se se muze nachazet komentar
i na nekolik radku.
*/
```

```
prikaz1; //neco se provede zde
prikaz2; //a neco jineho zde
```

Asymptote ignoruje prázdná místa tvořené z více mezer či zalomením řádků. Pozor však na psaní mezer mezi dvěma čísly, to je chápáno jako násobení. Následující příklady zdrojových kódů tak generují stejný výsledek.

```
//priklad1.asy
prikaz1;
prikaz2;
prikaz3;
```

```
//priklad2.asy
prikaz1; prikaz2; prikaz3;
```

```
//priklad3.asy
prikaz1;    prikaz2;
prikaz3;
```

2.2 Datové typy

Většina proměnných, s kterými v Asymptote budeme pracovat, musí mít předem deklarovaný typ. Asymptote nabízí několik předdefinovaných datových typů, běžně známých z jiných programovacích jazyků. Jsou to `int` pro celá čísla, `real` pro čísla v plovoucí desetinné čárce a `bool` pro pravdivostní hodnotu výrazu `true` nebo `false`. Dále nabízí například `pair`, který slouží pro uchovávání dvojic čísel typu `real` a je také chápán jako komplexní číslo. Následuje podrobnější seznam základních datových typů.

`int` – celé číslo, uloženo ve 64-bitovém formátu, mohou se v něm uchovávat celá čísla v rozmezí -9223372036854775808 až 9223372036854775805 (tj. -2^{63} až $2^{63}-3$). Tyto limity jsou uloženy proměnných v `intMin` a `intMax`.
`real` – číslo v plovoucí desetinné čárce, bývá implementováno jako nejpřesnější takový typ na dané architektuře. Používá `realDigits` význačných číslic a přesnost `realEpsilon`. Nejmenší a největší číslo tohoto typu je `realMin` a `realMax`.
`bool` – booleovský typ, může nabývat hodnot `true` a `false`.
`bool3` – rozšířený booleovský typ, kromě standardních `true` a `false` může nabývat hodnoty `default`.
`pair` – uspořádaná dvojice čísel typu `real`. Často bývá chápáno jako komplexní číslo. K jednotlivým složkám lze přistupovat přes atributy `x` a `y` přidáním tečky, tj. například `z.x`.
`triple` – uspořádaná trojice čísel typu `real`. Používáno pro body v prostoru či vektory. K jednotlivým složkám lze přistupovat přes atributy `x`, `y` a `z`.
`string` – posloupnost znaků zadaná ve dvojitých uvozovkách (text).

Deklaraci proměnné provedeme napsáním názvu typu před název proměnné a oddělíme mezerou, tj. například

```
nazev_typu1 promenna1;
nazev_typu2 promenna2;
nazev_typu3 promenna3;
```

deklaruje proměnné `promenna1`, `promenna2` a `promenna3` které mají postupně typy `nazev_typu1`, `nazev_typu2` a `nazev_typu3`. Součást deklarace může být přiřazení hodnoty (inicializace), které se provede operátorem =.

```
int n = 5;
real x = 0.01;
bool finished = false;
pair z = (1,2);
triple o = (0,1,0);
string label = "function f(x)";
```

Kromě uvedených typů existuje spousta dalších, uživatelsky definovaných. Existuje typ `struct` podobný jako v C, pro více informací viz [1, Documentation-6.8 Structures]). Obecně, pokud nějaký typ definuje atributy nebo metody, přistupujeme k nim pomocí znaku tečky.

```
struct Kruh {
  real r; //polomer

  real obsah(){
    return pi*r^2;
  }
}

Kruh k;
k.r = 5;

write("Polomer: ",k.r); //vypise atribut r
write("Obsah: ",k.obsah()); //spocita a vypise obsah
```

Následuje přehled některých pokročilých typů.

`frame` - plátno pro kreslení, pracuje v souřadnicích jazyka PostScript.
`picture` - podobný typ jako frame, avšak využívá vlastní souřadný systém. Pokud není uvedeno jinak, uživatel kreslí do plátna `currentpicture` typu `picture`.
`path` - představuje kubickou křivku.
`guide` - představuje také kubickou křivku, avšak uloženou jako posloupnost ne nutně všech kontrolních bodů. Takto zadaná křivka se do typu `path` převádí až těsně před vykreslením.

`pen` - pero pro kreslení, uchovává tloušťku, barvu, vzor apod. Pokud není uvedeno jinak, tak se kreslí perem `currentpen`.

Vypsaný seznam není úplný, k dispozici je mnoho užitečných typů, specifických pro danou problematiku a balík. Více typů bude probráno v kapitolách 3 a 4.

2.3 Pole

Deklaraci pole (konečné posloupnosti proměnných stejného typu) provedeme přidáním `[]` za název typu proměnné nebo za samotný název proměnné.

```
int[] pole;
real[] pole2 = {0.1, 0.2};
real pole3[];
```

Uvedený příklad vytváří tři pole, přitom `pole2` se přímo inicializuje s hodnotami 0,1 a 0,2. Pro přistupování k prvkům využijeme operátor `[]` s příslušným indexem uvnitř závorek. První prvek `pole2` bychom například získali příkazem `pole2[0]`. Pole má metody a atributy pro práci s ním, jako je mazání, vkládání apod.

`int length`	počet prvků pole
`void insert(int i ... T[] x)`	vloží prvek či pole prvků `x` na index `i`
`void delete(int i, int j=i)`	smaže prvky mezi indexy `i` a `j`
`T push(T x)`	vloží prvek na konec
`T pop()`	vyjme a vrátí poslední prvek

Tabulka 2.1: Základní atributy a metody polí

Můžeme vytvářet i tzv. anonymní pole, což jsou pole, která nemají žádný název. Vytvoří se vynecháním názvu proměnné a znaku „=“, a přidáním klíčového slova `new` před celý výraz. Následující kód by měl použití ozřejmit. Jedná se o definici čárkovaného pera, při které funkci předáváme pole dvou hodnot typu `real`.

```
pen dashed = linetype(new real [] {8 ,8});
```

2.4 Operátory

Pro práci s předdefinovanými typy nabízí Asymptote prakticky stejné operátory, jako jsou dostupné v C/C++. Základní přehled by měla ozřejmit následující ukázka.

```
int i = 0;
++i; //to same co i=i+1
i += 2; //stejne jako i=i+2
i = 12 % 5; //zbytek po deleni 5
real a = 2pi; //stejne jako 2*pi
```

Zmíněné operátory jsou často použitelné i pro jiné typy operandů, než jsou základní typy. Například je možné objektu typu `pen` nastavit tloušťku, jak ukazuje následující příklad (podrobněji o perech viz část 3.2).

```
pen mypen;
mypen += 2.0;
mypen += red;
```

2.5 Funkce

Při práci s proměnnými využíváme funkce. Jejich základní seznam lze nalézt na [1, Index]. Poněkud podrobnější seznam lze získat při spuštění Asymptote s parametrem `-l`. Ten však neobsahuje funkce z balíků. Proto bývá nejlepší prostudovat dokumentaci k jednotlivým balíkům, případně se podívat přímo do jejich kódu. V této části nebudeme popisovat jednotlivé funkce, ale rozebereme si jejich syntaxi. S konkrétními funkcemi se budeme setkávat napříč celou prací.

Nyní se podívejme, jak hlavičky funkcí vypadají a jak je interpretovat. Zde se Asymptote poměrně liší od jazyku C++ i Javy. Dovoleny jsou implicitní argumenty kdekoliv v hlavičce, explicitní uvedení názvu argumentu při volání funkce či umístění argumentu ne nutně tam, kde je v hlavičce uveden. Nejlépe si to ilustrujeme na příkladech.

Funkce `time` z balíku `math` spočítá n-tý průsečík křivky a svislé přímky.

```
real time(path g, real x, int n=0);
```

Samotná deklarace začíná typem návratové hodnoty. Jako typ se zde může uvést `void`, což značí, že funkce nevrací žádnou hodnotu. V našem případě funkce vrací typ `real`, tj. reálné číslo. Zde označuje čas na křivce, ve kterém se realizuje průsečík. Dále je uveden název funkce, tj. `time`, následovaný seznamem argumentů v závorkách. Argumenty (s výjimkou funkcí) jsou uvedeny v pořadí název typu, název argumentu a případně implicitní hodnota za znakem =. Pokud není uvedena implicitní hodnota, musíme argument vždy zadat. V tomto případě máme celočíselný nepovinný argument `n` (index průsečíku, který chceme spočítat).

Při volání funkce také můžeme explicitně zmínit název argumentu, který zadáváme. To umožňuje například zadávat argumenty v jiném pořadí, než uvádí hlavička. Následující tři volání jsou proto všechny ekvivalentní.

```
real t;
path p = (0,0)..(2,1)..(0,2);

t = time(p, 1);
t = time(p, 1, 0);
t = time(n=0,x=1,g=p);
```

Zde se ještě zastavme u tzv. přetížených funkcí. Tento koncept je převzatý z jazyka C++ a zajišťuje větší flexibilitu a přehlednost. Nemusíme mít funkce s různými jmény k tomu, abychom udělali stejný úkon, jen v jiném kontextu. Například pro vytváření ploch existuje spousta funkcí s názvem `surface()`. Každá se však používá s jinými typy argumentů, takže nenastává konflikt. Výše zmíněná funkce `time()` má také další variantu. Její hlavička vypadá následovně:

```
real time(path g, pair z, int n=0);
```

Slouží k počítaní průsečíků, tentokrát křivky a vodorovné přímky. Od předchozí deklarace se liší argumentem `z` typu `pair`. Odpovídající vodorovná přímka je pak určená bodem `(0,z.y)`. První složka `z` je zde nepodstatná, avšak autoři zde nemohli použít typ `real`, poněvadž by nastal konflikt s předchozí verzí funkce `time()`.

```
real t;
path p = (0,0)..(2,1)..(0,2);

t = time(p, 1); //1. varianta
t = time(p, (0,1)); //2. varianta
```

Funkce se také mohou nacházet v argumentech nebo být vráceny jinou funkcí. Jejich zápis mezi argumenty vypadá podobně, jako jejich deklarace, jen je vynechán název jednotlivých argumentů, tj. je uveden typ návratové hodnoty, název funkce a v závorce potom jednotlivé typy argumentů. Jako příklad uveďme následující variantu funkce `graph()` ze stejnojmenného balíku:

```
guide graph(picture pic=currentpicture, real f(real),
            real a, real b, int n=ngraph,
            real T(real)=identity,
            interpolate join=operator --);
```

Zde vidíme, že dva argumenty jsou funkce. První povinnou funkcí je zde `f`, která musí vracet typ `real`, a přebírat jeden argument typu `real`. Zřejmě se jedná o reálnou funkci jedné reálné proměnné, jíž chceme vykreslit graf. Další funkcí je zde `T` a má stejný návratový typ a typy argumentů jako `f`. Význam se samozřejmě liší, jedná se totiž o transformaci nezávislé proměnné a mezí (implicitně identita).

Pokud má funkce jako návratový typ jinou funkci, musí se předem pojmenovat typ takové funkce pomocí klíčového slova `typedef`. Nejlépe bude vše prezentovat opět na příkladu.

```
typedef real func_type(real);

func_type f(int n){
  real temp(real x){
    return n*sin(x/n);
  }
  return temp;
}
```

Zde klíčové slovo `typedef` říká, že pod názvem `func_type` budeme nyní rozumět typ takové funkce, která přebírá argument typu `real` a rovněž vrací hodnotu stejného typu.

Podobně jako u polí, i funkce můžeme vytvářet anonymní. Takové funkce nemají název, protože je využíváme pouze krátkodobě, většinou jako argument pro jinou funkci. Deklarace anonymní funkce se od obyčejné liší pouze tím, že jí chybí název a před celý výraz je přidáno klíčové slovo `new`.

```
//"klasicky" zpusob
real f(real x){
  return sin(x);
}

draw(graph(f, -2pi, 2pi));
```

```
//s vyuzitim anonymni funkce
draw(graph(new real(real x){ return sin(x);}, -2pi, 2pi));
```

2.6 Cykly

Asymptote má syntaxi pro cykly `for`, `while` a `do` shodnou s C/C++ či Javou, od druhého jmenovaného si navíc propůjčuje syntaxi pro procházení prvků pole. Následující ukázka kódu prezentuje výpis čísel od 0 do 4 pomocí různých cyklů.

```
//ukazka for, while, do a foreach
for(int i = 0; i < 5; ++i){
  write(i);
```

```
}

int i = 0;
while(i < 5){
  write(i);
  ++i;
}

int i = 0;
do{
  write(i);
  ++i;
}while(i < 5);

int[] pole = {0,1,2,3,4};
for(int i : pole){
  write(i);
}
```

2.7 Balíky

Asymptote nabízí spoustu předpřipravených balíků pro různé účely. Při jejich načtení máme k dispozici celou řadu užitečných funkcí. Máme tak například balík `graph` pro snadnou tvorbu grafů funkcí, `animation` pro vytváření animací nebo třeba `MetaPost`, obsahující funkce kompatibilní s tímto systémem. Asymptote však není prostředek pouze pro vykreslování grafiky, nabízí také různé matematické funkce, například pro řešení obyčejných diferenciálních rovnic (balík `ode`), řešení úlohy lineárního programování simplexovou metodou (balík `simplex`), či některé obecnější funkce (balík `math`). Dále si uživatel může balíky sám vytvářet, jsou to obyčejné soubory formátu `.asy` (příklad viz část 3.7.2). Jejich vkládání se pak provádí příkazem `import`. Například

```
import three;
```

načte balík pro práci s vektory ve 3D (uložený v souboru `three.asy`). V této práci si předvedeme základní balíky pro práci ve 2D i ve 3D.

Občas je nutné načíst nejen Asymptote balíky, ale také LaTeXové balíky. K tomu slouží funkce `usepackage()`:

```
void usepackage(string s, string options="");
```

```
usepackage("amssymb");
```

Využijeme ji tehdy, když v popiskách používáme konstrukce, které i v samotném LaTeXu vyžadují nějaký balík.

2.8 Matematické konstanty a funkce

Zde se již nebudeme zabývat syntaxí, ale uvedeme si základní matematické funkce. Jejich význam však není nijak svázán s tím, zda pracujeme ve 2D nebo 3D, proto je tato část umístěna zde. Pro některé uvedené funkce a proměnné je potřeba načíst balík `math` příkazem:

```
import math;
```

Co se týče číselných konstant, používá Asymptote následující:

```
pi = 3.14159265358979;
I = (0,1); //komplexni jednotka
```

Z elementárních funkcí jsou k dispozici:

```
real sin(real x); //goniometricke funkce
real cos(real x);
real tan(real x);
real asin(real x); //cyklometricke funkce
real acos(real x);
real atan(real x);
real exp(real x); //exponencialni funkce
real log(real x); //prirozeny logaritmus
real log10(real x);
real sqrt(real x); //druha odmocnina
real cbrt(real x); //treti odmocnina
```

Dále jsou definovány také hyperbolické funkce s jejich inverzemi:

```
real sinh(real x); //hyperbolicke funkce
real cosh(real x);
real tanh(real x);
real asinh(real x);
real acosh(real x);
real atanh(real x);
```

K uvedeným funkcím `sin`, `cos`, `tan`, `exp`, `log` a `sqrt` existují také jejich komplexní varianty, jen místo argumentu `real` používají typ `pair`.

Výpočet absolutní hodnoty obstará funkce:

```
real abs(real x); //absolutni hodnota
```

Tato funkce má také varianty pro argumenty typu `pair` a `triple`, vracející velikost příslušných vektorů.

Pro převod mezi stupni a radiány lze využít:

```
real degrees(real radians);
real radians(real degrees);
```

Pro zaokrouhlování jsou k dispozici:

```
int floor(real x);
int ceil(real x);
int round(real);
```

Faktoriál a kombinační čísla lze vytvořit funkcemi:

```
int factorial(int n);
int choose(int n, int k);
```

Náhodná čísla můžeme generovat použitím:

```
void srand(int seed); //(nepovinna) inicializace seminkem
int rand(); //nahodne cele cislo v~intervalu [0,randMax]
real unitrand(); //nahodne cislo v~intervalu [0,1]
```

Kapitola 3

Práce ve 2D

3.1 Plátna

Vše, co v Asymptote kreslíme, se odehrává na nějakém plátně. Pro reprezentaci plátna existují dva typy. Jsou jimi `frame` a `picture`. Liší se od sebe tím, že `frame` interpretuje všechny délky v jednotkách bp jazyka PostScript, zatímco `picture` umožňuje definovat vlastní jednotky. Ty se pak samy převádí na jednotky jazyka PostScript. Většinou se používá právě typ `picture`.

3.1.1 Jednotky a typ `picture`

Pokud žádné jednotky nespecifikujeme, všechna čísla se budou chápat v jednotkách bigpoint jazyka PostScript (1bp $= 1/72$ palce). Praktičtější je však nastavit vlastní jednotky v rámci plátna, které je typu `picture`. Slouží k tomu funkce `unitsize`.

```
void unitsize(picture pic=currentpicture,
              real x, real y=x);
```

Pokud nespecifikujeme plátno `pic`, použije se `currentpicture`. Argumenty `x` a `y` určují, jaké jednotky se použijí ve směru osy x, případně y. Můžeme tak například napsat:

```
unitsize(1cm);
//stejne jako
//unitsize(1cm,1cm)
```

Vše, co potom vykreslíme v implicitním plátně `currentpicture`, bude v jednotkách centimetrů. Kromě `cm` lze také použít `mm`, `inches`, `bp` a `pt`. Další funkce se zdá být v praxi užitečnější, nastaví totiž výstupní rozměry pro daný obrázek.

```
void size(picture pic=currentpicture, real x, real y=x,
          bool keepAspect=Aspect);
```

Výsledný obrázek nebude mít šířku přesahující `x` a výšku přesahující `y`. Pokud se za proměnnou `x` příp. `y` předá 0, nebude v daném směru kladeno žádné omezení. Pokud se za proměnnou `keepAspect` použije hodnota `Aspect` nebo `true`, pak bude zachován poměr stran a obrázek se vykreslí tak, aby šířka ani výška nepřesáhly zadané hranice. Pokud se použije hodnota `IgnoreAspect` nebo `false`, poměr stran se upraví tak, aby šířka i výška obrázku odpovídala zadaným `x` a `y`.

```
void draw(picture pic=currentpicture, Label L="", path g,
          align align=NoAlign, pen p=currentpen,
          arrowbar arrow=None, arrowbar bar=None,
          margin margin=NoMargin, Label legend="",
          marker marker=nomarker);
```

Do obrázku `pic` vykreslí křivku předanou v parametru `g`. Přitom je možno nastavit pero `p`, případně popisek `L`. Jde o základní funkci pro vykreslování.

Jednotky jsou implementovány jako předdefinované konstanty typu `real` a jsou to násobky bigpointu. Například hodnota konstanty `cm` je přibližně 2,83, poněvadž 1cm $\approx$ 2,83bp. Z tohoto důvodu je při použití `unitsize` nežádoucí, uvádět kdekoliv jinde v kódu jednotky (s výjimkou míst, která jsou na jednotkách plátna nezávislé). Z povahy implementace by totiž takto vynásobená čísla s největší pravděpodobností neodpovídala zamýšlené délce. Ze stejného důvodu je nevhodné pro názvy proměnných používat jednotky.

3.2 Pera

Pera hrají velmi důležitou roli při kreslení. Pomocí nich uživatel nastaví tloušťku, barvu, velikost písma a mnoho dalších. Pera mají typ `pen` a nejčastěji se používají jako argumenty 4 základních funkcí pro vykreslování `draw()`, `fill()`, `clip()` a `label()`. Pokud při kreslení neuvedeme žádné pero, je použito `currentpen`. Můžeme také globálně nastavit, aby `currentpen` mělo požadované vlastnosti.

V dalších částech si popíšeme, jak se vytvářejí pera určitých vlastností. Takto vytvořená pera se dají kombinovat binárním operátorem `+`, případně operátorem `*` pro násobení hodnot barevných složek číslem.

3.2.1 Barvy

Pera určité barvy můžeme vytvářet na základě RGB i CMYK příslušnými funkcemi.

```
pen rgb(real r, real g, real b);
```

```
pen cmyk(real c, real m, real y, real k);
```

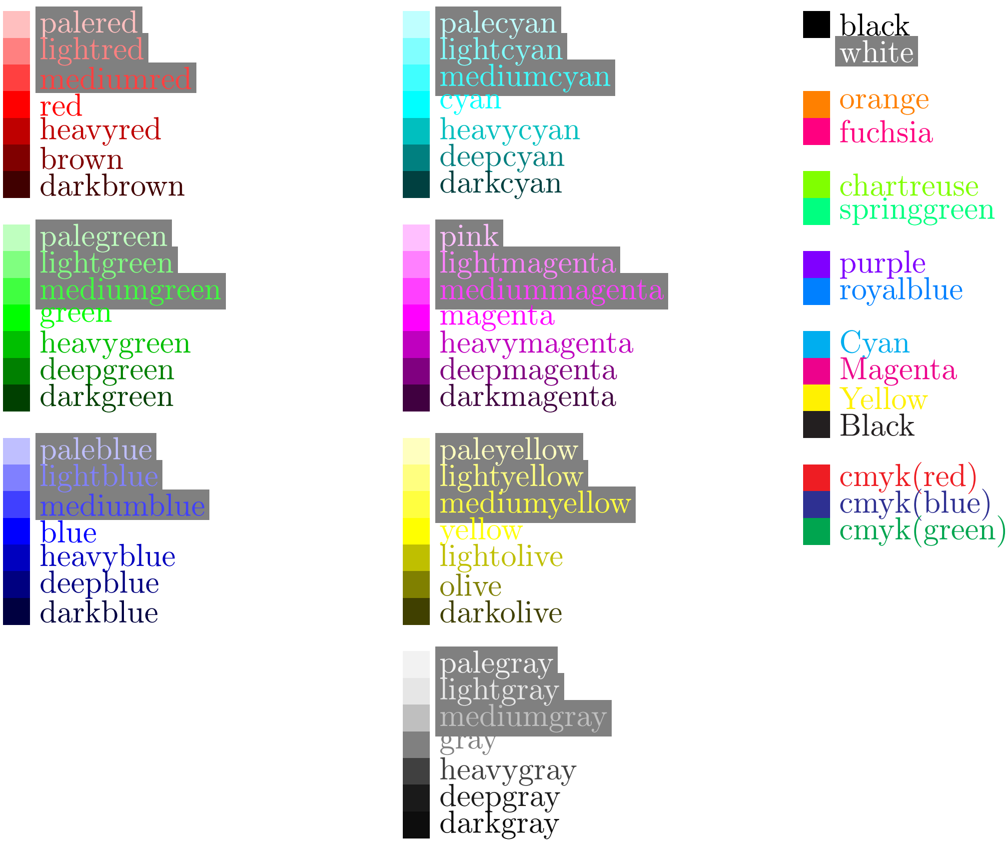
Argumenty `r`, `g`, `b` případně `c`, `m`, `y`, `k` leží v intervalu [0,1]. Pokud chceme definovat například pero s červenou barvou, můžeme napsat:

```
pen cervena = rgb(1,0,0);
```

Většinou však využijeme předpřipravené palety, případně namícháme barvy pomocí operátoru `+`. Například

```
pen p = red+green;
```

vytvoří žlutou barvu. Následuje přehled předdefinovaných barev.



Obr. 3.1: Barevná paleta

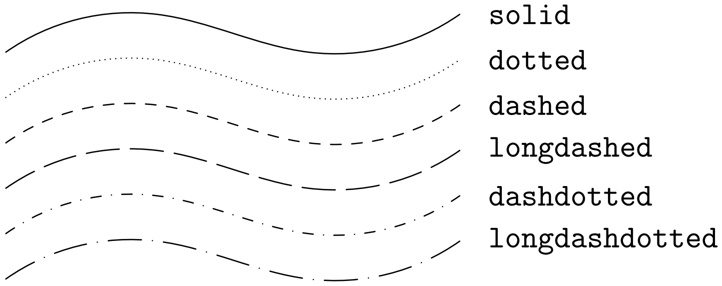
3.2.2 Typy čar

Typy čar jsou v Asymptote určeny posloupností čísel typu `real` (polem) spolu s dodatečnými parametry. Uveďme si nejdříve hlavičku funkce pro sestrojení pera s určitým typem čáry. Potom popíšeme význam jednotlivých argumentů.

```
pen linetype(real[] a, real offset=0,
             bool scale=true, bool adjust=true);
```

Pole čísel `a` odpovídá délkám po sobě jdoucích úseků, kde první číslo udává délku viditelného úseku, druhé číslo pak délku následujícího neviditelného úseku, třetí číslo opět následujícího viditelného atd. Zde 0 odpovídá tečce. Argument `offset` určuje počáteční posunutí. Pomocí `scale` říkáme, jestli se zadané délky úseků mají chápat jako násobky tloušťky čáry (pro `scale=true`). Argument `adjust` zajistí, aby se délky mezer mezi úseky přizpůsobovaly délce dané křivky, takže se nestane, aby křivka skončila mezerou nebo jen částečnou čárkou. Důležité také je, že zadané délky nejsou nijak svázané s jednotkami plátna a jsou tak v jednotkách `bp` jazyka PostScript.

Podobně jako u barev, i zde Asymptote nabízí předefinované konstanty pro nejpoužívanější typy čar. Implicitně se využívá typ `solid`.



Obr. 3.2: Typy čar

Pro názornost původní definice se ještě podívejme, jak je definován čárkovaný (`dashed`) a čerchovaný (`dashdotted`) typ čar.

```
pen dashed=linetype(new real[] {8,8});
pen dashdotted=linetype(new real[] {8,8,0,8});
```

U obou typů je argument `scale` implicitně roven `true`, a proto se zadané délky budou chovat jako násobky aktuální tloušťky pera `linewidth(p)`. Čárkovaná čára je tedy definovaná tak, že vždy úsek délky $8 \cdot$ `linewidth(p)` je viditelný a následuje ho prázdný úsek délky $8 \cdot$ `linewidth(p)`. Podobně je to s čerchovanou čárou, kde 0 znamená tečku.

3.3 Křivky

Základním útvarem, který budeme v Asymptote konstruovat, přímo či nepřímo, jsou křivky. Všechny úsečky, grafy funkcí nebo třeba i popisky jsou uloženy jako segmenty kubických křivek. Pro křivky v rovině má Asymptote typy `guide` a `path`. Typ `guide` pouze uchovává posloupnost bodů, ale nemá ještě dopočítány všechny údaje (je vhodnější ve fázi sestrojování křivky). Typ `guide` se těsně před vykreslením převádí na typ `path`, ve kterém už jsou dopočítány všechny kontrolní body.

Křivku tvoříme navazováním bodů vhodnými operátory, které mezi nimi určují způsob interpolace. Pokud má být křivka uzavřená, jako poslední bod uvedeme `cycle`. Pro lineární interpolaci (spojení úsečkou) používáme `operator --`. Například jednotkový čtverec můžeme vytvořit takto:

```
path ctverec = (0,0)--(1,0)--(1,1)--(0,1)--cycle;
```

Jednotkový čtverec je mimochodem v Asymptote předdefinovaný právě tímto způsobem a to v proměnné `unitsquare`.

Pro interpolaci Bézierovou kubikou používáme `operator ..`. Ten se dá použít s různými parametry, popišme si ho proto podrobněji.

Bézierova kubika je zadaná celkem čtyřmi kontrolními body. Vždy budeme muset zadat krajní body, kterými prochází. Zbytek může Asymptote dopočítat. Pokud chceme zadat i kontrolní body, použijeme zápis:

```
a .. controls c0 and c1 .. b
```

V případě, že nezadáváme kontrolní body přímo, vypadá zápis s parametry takto:

```
a{dir1} .. tension t1 and t2 .. {dir2}b
```

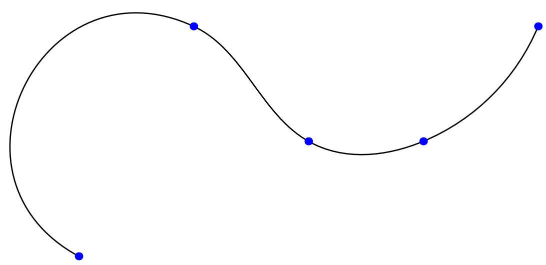
Parametry `dir1` a `dir2` udávají směry tečných vektorů segmentu v prvním a druhém krajním bodě (jsou typu `pair`). Parametry `t1` a `t2` určují míru zakřivení v podobě reálného čísla většího nebo rovného 0,75. Čím větší tato hodnota je, tím víc se tvar blíží přímce. Pokud nejsou zadány, volí se hodnota 1. Žádný z těchto parametrů však zadávat nemusíme, nezadané směry a výsledné kontrolní body se určí pomocí tzv. Hobbyho algoritmu.

Pro samotné vykreslování křivek slouží funkce `draw()`.

```
void draw(picture pic=currentpicture, Label L="",
          path g, align align=NoAlign,
          pen p=currentpen, arrowbar arrow=None,
          arrowbar bar=None, margin margin=NoMargin,
          Label legend="", marker marker=nomarker);
```

Tato varianta vykreslí křivku `g` do plátna `pic`. Před popisem dalších argumentů si uveďme jednoduchý příklad.

```
size(200,0);

pair[] z = {(0,0),(1,2),(2,1),(3,1),(4,2)};

path p =z[0]..z[1]..z[2]..z[3]..z[4];

draw(p);
dot(z,blue);
```

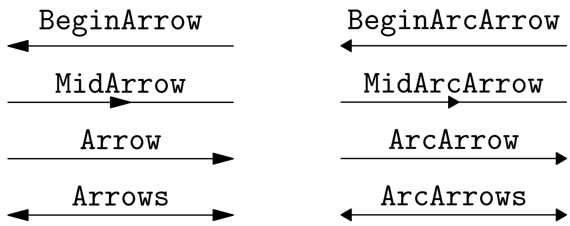


Obr. 3.3: Vykreslení typu `path`

Mezi další argumenty patří popisek `L` a jeho směr zarovnání `align`. Dále `arrow` umožňuje přidat na křivku šipku. Možné hodnoty jsou vidět na obr. 3.4. Každá z těchto šipek jde navíc volat jako stejnojmenná funkce s argumenty určující velikost, tvar apod. Pro více informací se doporučuji podívat do balíku `plain_arrows`.

```
picture left, right;

size(left,3cm);
size(right,3cm);

path p = (0,0)--(2,0);

pair align = N;

draw(left,"\tt BeginArrow",p,align,BeginArrow);
draw(left,"\tt MidArrow",shift(0,-0.5)*p,align,MidArrow);
draw(left,"\tt Arrow",shift(0,-1)*p,align,Arrow);
draw(left,"\tt Arrows",shift(0,-1.5)*p,align,Arrows);

draw(right,"\tt BeginArcArrow",p,align,BeginArcArrow);
draw(right,"\tt MidArcArrow",shift(0,-0.5)*p,align,
   MidArcArrow);
draw(right,"\tt ArcArrow",shift(0,-1)*p,align,ArcArrow);
```

```
draw(right,"\tt ArcArrows",shift(0,-1.5)*p,align,ArcArrows);

add(left.fit(),(0,0),20W);
add(right.fit(),(0,0),20E);
```



Obr. 3.4: Šipky

Použitím `bar` můžeme přidat příčku na konci křivky:

```
size(3cm);

path p = (0,0)--(2,0);

pair align = N;

draw("\tt BeginBar",p,align,bar=BeginBar);
draw("\tt Bar",shift(0,-0.5)*p,align,bar=Bar);
draw("\tt Bars",shift(0,-1)*p,align,bar=Bars);
```

Obr. 3.5: Příčky

Jelikož křivky jsou orientované a často určují nějakou oblast (například pro vyplnění), hodí se nám funkce `reverse` pro změnu parametrizace na opačný směr.

```
path reverse(path p);
```

Další funkce (pro počítání průsečíků apod.) lze nalézt na [1, Documentation-6.2 Paths and guides].

3.4 Vyplňování

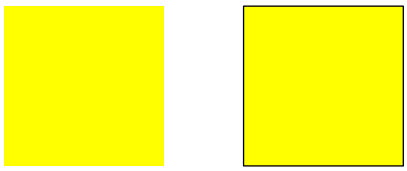
Uzavřené křivky můžeme vyplnit funkcí `fill()`, případně `filldraw()`.

```
void fill(picture pic=currentpicture, path g,
          pen p=currentpen);
```

```
void filldraw(picture pic=currentpicture, path g,
              pen fillpen=currentpen,
              pen drawpen=currentpen);
```

Liší se od sebe pouze tím, že druhá funkce také vykreslí obrys, viz obr. 3.6.

```
size(150);

fill(unitsquare,yellow);
filldraw(shift(1.5,0)*unitsquare,yellow,black);
```

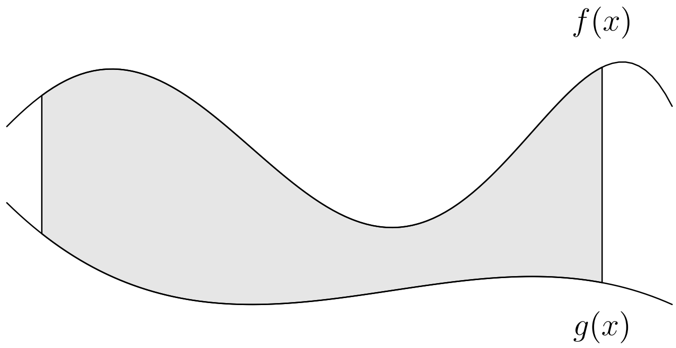


Obr. 3.6: Vyplnění funkcemi `fill()` a `filldraw()`

Funkci můžeme předat i křivky, které jsou vytvořené funkcemi pro tvorbu grafů. Následující ukázka ilustruje, jak vyplnit oblast mezi grafy funkcí $f(x)$ a $g(x)$ (podrobněji o grafech viz část 3.7).

```
import graph;

size(250,0);

real f(real x){
  return -1/40*x^5+1/4*x^4-2/3*x^3+9/8*x+2;
}

real g(real x){
  return -1/20*x^3+9/20*x^2-6/5*x+2;
}

filldraw(graph(f,0.5,4.5)--graph(g,4.5,0.5)--cycle,
         lightgray);

draw(graph(f,0.25,5),black);
draw(graph(g,0.25,5),black);
```

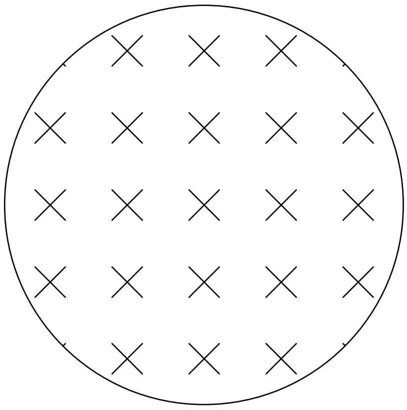
```
label("$f(x)$",(4.5,f(4.5)),3N);
label("$g(x)$",(4.5,g(4.5)),3S);
```



Obr. 3.7: Vyplnění funkcí `fill`

Pero nemusí obsahovat jen základní informace jako barvu, ale může definovat i vzor pro vyplnění, kterým může být prakticky libovolný obrázek.

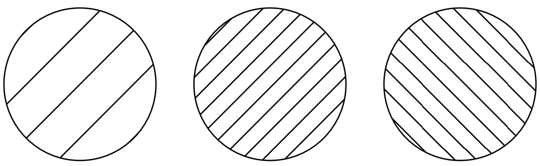
```
import patterns;

size(0,150);

//krizek
picture custom;
real h = 2mm;
path[] cross = (-1,-1)--(1,1)^^(1,-1)--(-1,1);
draw(custom,scale(h)*cross);

//pridani do vzoru
add("custompattern",custom,(3h,3h));

filldraw(unitcircle,pattern("custompattern"));
```



Obr. 3.8: Vyplnění vlastním vzorem

V uvedeném příkladu jsme do vlastního plátna nakreslili křížek. Použitím funkce `add` jsme ho pak přidali do seznamu vzorů. Funkce `pattern` potom vytvořila příslušné pero, se kterým byla vyplněna kružnice. Pro praktické použití máme v balíku `patterns` k dispozici funkce pro vytváření obrázků, které můžeme použít pro různé typy šrafování.

```
import patterns;

size(200,0);

add("hatch",hatch());
add("hatch2",hatch(2mm));
add("hatchback2",hatch(2mm,NW));

filldraw(unitcircle,pattern("hatch"));
filldraw(shift(2.5,0)*unitcircle,pattern("hatch2"));
filldraw(shift(5,0)*unitcircle,pattern("hatchback2"));
```



Obr. 3.9: Šrafování

3.5 Ořezávání

```
void clip(picture pic=currentpicture, path g, stroke=false,
          pen fillrule=currentpen);
```

Funkcí `clip` můžeme oříznout obsah plátna `pic` křivkou `g`. Pro matematickou grafiku jsou však zajímavější následující dvě funkce.

```
void xlimits(picture pic=currentpicture, real min=-infinity,
             real max=infinity, bool crop=NoCrop);
```

```
void ylimits(picture pic=currentpicture, real min=-infinity,
             real max=infinity, bool crop=NoCrop);
```

```
void crop(picture pic=currentpicture);
```

Tyto funkce nastaví meze obrázku ve směrech os x a y a následné volání funkce `crop` ořeže existující křivky na plátně podle zadaných mezí. To je praktické zejména při kreslení grafů, kdy chceme, aby takový graf nepřesahoval zadané meze. Použití lze vidět např. na obr. 3.13 v části 3.7.2.

3.6 Transformace

Pro účely afinních transformací existuje typ `transform`. Proměnnou tohoto typu můžeme vynásobit zleva libovolnou proměnnou typu `pair`, `guide`, `path`, `pen`, `string`, `frame` nebo `picture`. Výsledně tak obdržíme proměnnou stejného typu, po provedení dané transformace. Popišme si, co transformace pro konkrétní typy znamená.
`pair` - transformace se aplikuje na bod v rovině
`guide`, `path` - transformace se aplikuje na všechny body daných křivek včetně kontrolních bodů (Bézierovy křivky jsou však invariantní vůči afinním transformacím)
`string` - převede se na typ `Label`, což není nic jiného než křivka a na tu se následně aplikuje transformace
`frame`, `picture` - transformace se aplikuje na obsah plátna (těsně před vykreslením)

Uživatel si samozřejmě může definovat vlastní transformace. Obecná afinní transformace

$$t(x) = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} x + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{3.1}$$

se vytvoří příkazem:

```
transform t = (b1,b2,a11,a12,a21,a22);
```

Pro jednodušší práci jsou ty nejpoužívanější transformace předdefinovány. Následuje seznam a popis funkcí, které vytváří a vrací příslušné transformace.

```
transform identity();
```

Identická transformace. Slouží víceméně jen proto, aby se dala použít jako implicitní transformace.

```
transform shift(pair z);
transform shift(real x, real y);
```

Translace/posunutí určené buďto vektorem `z`, a nebo jednotlivými složkami `x`, `y` zvlášť.

```
transform rotate(real angle, pair z=(0,0));
```

Otočení o úhel `angle` (ve stupních) kolem bodu `z`.

```
transform scale(real s);
transform scale(real x, real y);
transform xscale(real x);
transform yscale(real y);
```

Změna měřítka, opět je možno zadat více způsoby. Funkce `scale()` mění měřítko ve směru obou os. Zbylé funkce umožňují měnit měřítka pro obě osy zvlášť.

```
transform reflect(pair a, pair b);
```

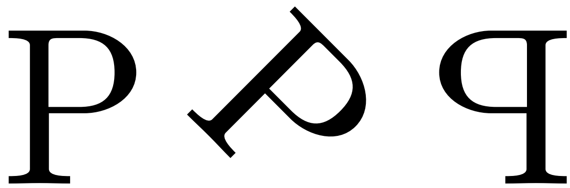
Převrácení kolem přímky určené body `a` a `b`.

K dané transformaci `t` můžeme získat inverzní transformaci pomocí funkce `inverse`:

```
transform inverse(transform t);
```

Transformace můžeme skládat, a to pomocí operátoru násobení (`*`). Na závěr uveďme ilustrační příklad.

```
size(0,200);

Label l = scale(7)*"P";

//pismeno "P"
label(l);

//otocene "P" o -45 stupnu
label(shift(80,0)*rotate(-45)*l);

//prevracene "P"
label(shift(160,0)*reflect((0,0),(0,1))*l);
```



Obr. 3.10: Transformace

3.7 Grafy funkcí

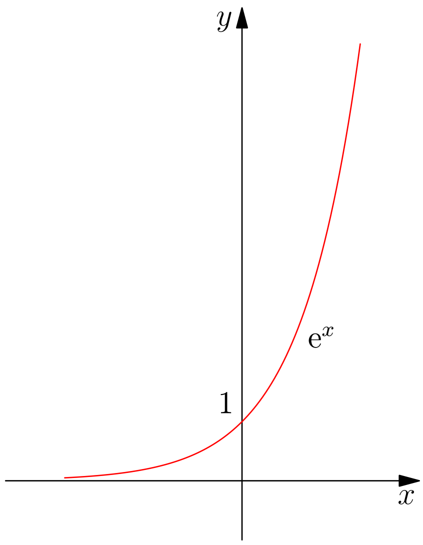
3.7.1 Funkce $f(x)$

Při vykreslování grafů funkcí budeme využívat balík `graph`. Pro vykreslení reálné funkce v explicitním tvaru $y = f(x)$ využijeme následující variantu funkce `graph()`:

```
guide graph(picture pic=currentpicture, real f(real), real a,
            real b, int n=ngraph, real T(real)=identity,
            interpolate join=operator --);
```

Funkce spočítá funkční hodnoty podle zadaných parametrů a vrátí vytvořenou křivku. Argument `f` musí být funkce mající jeden argument typu `real` a rovněž vracející hodnotu stejného typu (tj. odpovídající reálná funkce jedné reálné proměnné). Argumenty `a` a `b` určují meze na ose x. Mezi nepovinnými argumenty můžeme zadat plátno `pic`, případně dělení `n`. Předveďme si použití na vykreslení grafu funkce $f(x) = \mathrm{e}^x$.

```
import graph;

size(200);

real f(real x){
  return exp(x);
}

draw(graph(f,-3,2),red);

xaxis("$x$",-4,3,Arrow);
yaxis("$y$",-1,8,Arrow);

labely("$1$",1,NW);
label("$\mathrm{e}^x$",(1,f(1)),SE);
```



Obr. 3.11: Funkce $f(x) = \mathrm{e}^x$

3.7.2 Nespojité funkce

V případě nespojitých funkcí (příp. funkcí nedefinovaných na celém zadaném intervalu) je situace komplikovanější. Asymptote totiž při počítání funkčních hodnot v bodech, kde není definována funkční hodnota, vyvolá chybu a graf se nevykreslí. U nespojitých funkcí naopak nemá informaci o tom, že v daném místě nemá části grafu spojovat. To se dá vyřešit několika způsoby.

Nevhodná metoda, pro řešení prvního zmíněného problému, je dodefinovat funkční hodnoty rovny například nule, protože pak dostáváme jiný graf. Trochu lepším řešením je vykreslovat části grafu zvlášť, a to na intervalech, kde je funkce definována. Řešením, které umožňuje vypořádat se i s druhým zmíněným problémem, je argument `cond` ve funkci `graph()`. Jeho hlavička je očekávána ve tvaru:

```
bool3 cond(real x);
```

Tato, námi zadaná funkce, umožňuje sdělovat funkci `graph()`, které hodnoty na ose x smí či nesmí vyhodnocovat. To vše navíc tak, že bude vědět, kdy začíná další větev grafu, takže nedojde k jejich propojení. Pro lepší pochopení následujícího příkladu (a pro opětovnou využitelnost) si definujeme vlastní jednoduchý balík `discont`, který za nás provede výše popsaný postup. My pouze specifikujeme seznam intervalů, na kterých není funkce spojitá nebo není definována (tj. intervaly, kde její graf nechceme vykreslovat).

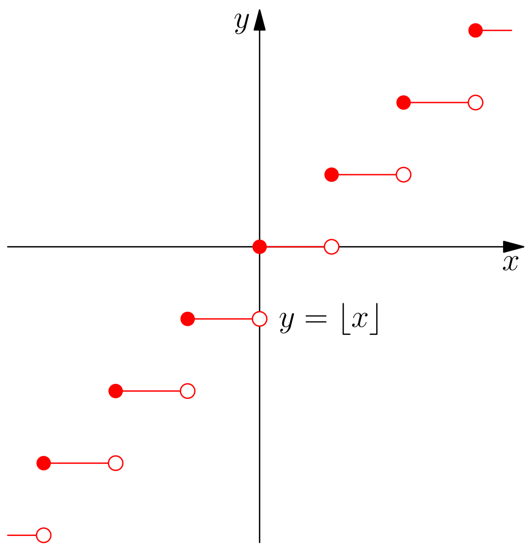
```
//discont.asy
//verejne promenne pro nastaveni
```

```
pair[] discontIntervals;
int discontLastIndex = 0;

//funkce pouzitelna jako argument cond v graph()
bool3 discontCond(real x){
  if (discontLastIndex >= discontIntervals.length){
    //za poslednim intervalem nespojitosti
    return true;
  }

  if (x <= discontIntervals[discontLastIndex].x){
    //stale vlevo od pristiho intervalu nespojitosti
    return true;
  } else if (x >= discontIntervals[discontLastIndex].y) {
    //interval nespojitosti prekrocen
    ++discontLastIndex;
    return default;
  }

  //v okoli bodu nespojitosti
  return false;
}
```

Balík umístíme do stejné složky s hlavním souborem, příp. do složky s ostatními balíky Asymptote. V hlavním souboru pak balík načteme, nastavíme příslušný seznam intervalů `discontIntervals` a nakonec funkci `graph()` předáme jako argument funkci `discontCond` z našeho balíku. Seznam je implementován jako pole hodnot typu `pair`, kde jednotlivé složky značí krajní body daného intervalu (bod můžeme zadat jako interval s totožnými krajními body). V případě více grafů navíc před každým kreslením vynulujeme pozici aktuálního intervalu a to příkazem `discontLastIndex = 0`. Ukažme si použití na funkci $y = \lfloor x \rfloor$.

```
import graph;
import discont;

size(200);

real f(real x){
  return floor(x);
}

//intervaly kolem hodnot -4,-3,...,4
real eps = 0.01;
for(int i=-4;i<=4;++i){
  discontIntervals.push((i-eps,i+eps));
}
```

```
draw(graph(f,-3.5,3.5,n=2000,discontCond),red);

path circle = scale(0.1)*unitcircle;
for(int i=-3;i<=3;++i){
  fill(shift((i,f(i)))*circle,red);
  filldraw(shift((i,f(i)-1))*circle,white,red);
}

xaxis("$x$",Arrow);
yaxis("$y$",Arrow);

label("$y=\lfloor x\rfloor$",(0,-1),2E);
```



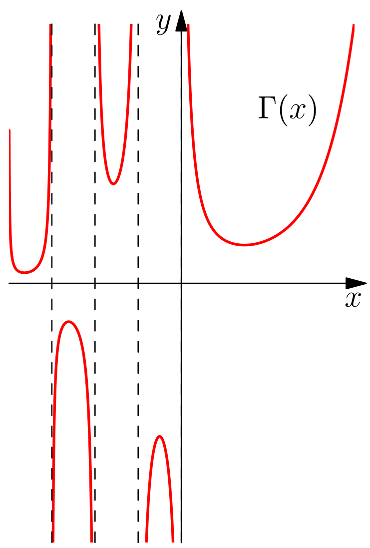
Obr. 3.12: Graf funkce $y = \lfloor x \rfloor$

V seznamu jsme tedy jako intervaly nespojitosti zadali $(n-\epsilon, n+\epsilon)$ pro $n = -4, -3, \ldots, 4$ a $\epsilon = 0{,}01$. V těch se při sestrojování grafu nevyhodnocovali funkční hodnoty a zároveň byly okolní větve správně odděleny. Pro použití u jiné funkce stačí pouze nahradit pole `discontIntervals` vlastními intervaly či body. Na uživateli je dodržet, aby intervaly byly disjunktní a seřazené dle krajních hodnot od nejmenší po největší. Následující příklad ukazuje použití na gamma funkci

$$y = \Gamma(x) = \int_0^\infty t^{x-1} \mathrm{e}^{-t} \mathrm{d}t, \tag{3.2}$$

která v bodech $-4, -3, \ldots, 0$ není definována.

```
import graph;
import discont; //vlastni balik

size(0,200,IgnoreAspect);

real f(real x){
  return 1/x;
}

//intervaly kolem hodnot -4,-3,-2,-1 a 0
real eps = 0.01;
for(int i=-4;i<=0;++i){
  discontIntervals.push((i-eps,i+eps));
}

draw(graph(gamma,-4,4,n=2000,discontCond),red+1.0);

ylimits(-6,6);
crop();
xaxis("$x$",Arrow);
yaxis("$y$",Arrow);

for(int i=-3;i<=0;++i){
  xequals(i,dashed);
}

label("$\Gamma(x)$",(3.5,gamma(3.5)),2NW);
```



Obr. 3.13: Graf funkce $y = \Gamma(x)$

3.7.3 Křivka zadaná parametricky

Graf křivky $f(t) = (x(t), y(t))$ vytvoříme následující verzí funkce `graph()`:

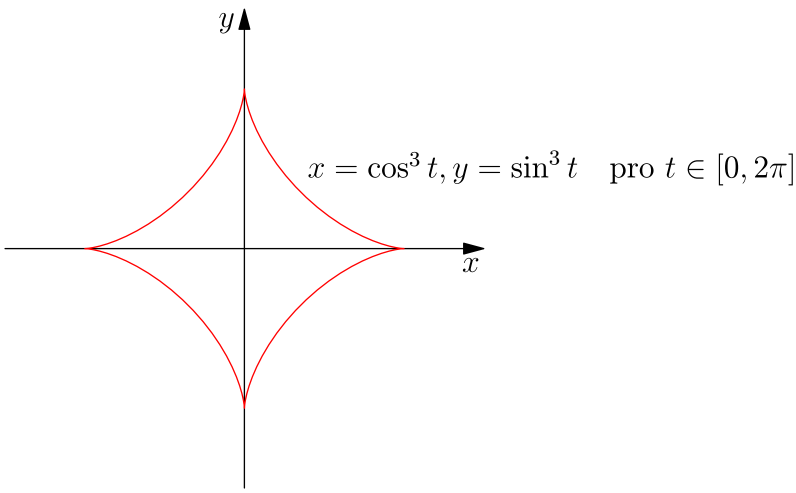
```
guide graph(picture pic=currentpicture, real x(real),
            real y(real), real a, real b,
            int n=ngraph, real T(real)=identity,
            interpolate join=operator --);
```

Od předešlé verze `graph()` se liší jen možností zadat dvě reálné funkce. Například graf asteroidy, zadané parametricky

$$x = \cos^3 t, y = \sin^3 t \quad \text{pro } t \in [0, 2\pi], \tag{3.3}$$

můžeme vykreslit takto:

```
import graph;

size(0,180);

real x(real t) {return cos(t)^3;}
real y(real t) {return sin(t)^3;}

label("$x=\cos ^3 t, y=\sin ^3 t
       \quad\mbox{pro $t\in [0,2\pi]$}$",
       (x(pi/4),y(pi/4)),NE);

draw(graph(x,y,0,2pi),red);
xaxis("$x$",-1.5,1.5,Arrow);
yaxis("$y$",-1.5,1.5,Arrow);
```



Obr. 3.14: Parametrická křivka

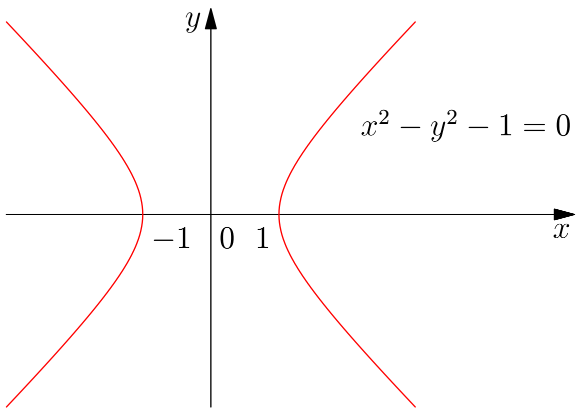
3.7.4 Implicitní funkce

Graf funkce v implicitním tvaru $F(x, y) = 0$ můžeme vykreslit pomocí funkce `contour` ze stejnojmenného balíku. Ta slouží pro vytváření vrstevnic ve tvaru $F(x, y) = \mathrm{C}$.

```
guide[][] contour(real F(real, real), pair a, pair b,
                  real[] c, int nx=ngraph, int ny=nx,
                  interpolate join=operator --,
                  int subsample=1);
```

V poli `c` předáváme množinu hodnot pro C, pro které chceme vytvořit vrstevnice $F(x, y) = \mathrm{C}$, v rozsazích hodnot od `a` do `b`.

```
import contour;
import graph;

size(0,150);

real f(real x, real y){
  return x^2-y^2-1;
}

draw(contour(f,(-3,-3),(3,3),new real[]{0},join=operator ..),
   red);

xaxis("$x$",Arrow);
yaxis("$y$",Arrow);
label("$x^2-y^2-1=0$",(2,sqrt(3)),2SE);
labelx(-1,SE);
labelx(0,SE);
labelx(1,SW);
```



Obr. 3.15: Implicitní funkce

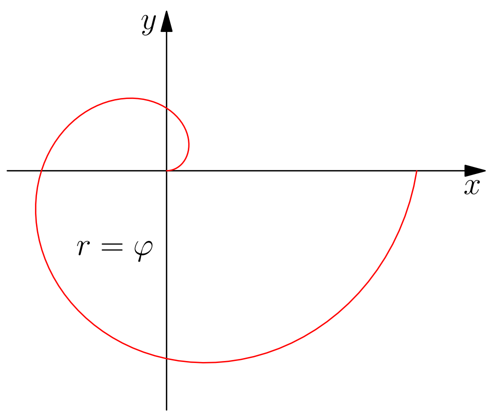
3.7.5 Křivka zadaná v polárních souřadnicích

Křivku v polárních souřadnicích, zadanou ve tvaru $r = r(\varphi)$ pro $\varphi \in [a, b]$, můžeme vytvořit pomocí funkce `polargraph` z balíku `graph`.

```
guide polargraph(picture pic=currentpicture,
                 real r(real), real a,
                 real b, int n=ngraph,
                 interpolate join=operator --);
```

Evolventu, zadanou v polárních souřadnicích jako $r = \varphi$, můžeme pak vytvořit například takto:

```
import graph;

size(0,150);

real r(real phi){
  return phi;
}

draw(polargraph(r,0,2pi),red);

xaxis("$x$",-4,8,Arrow);
yaxis("$y$",-6,4,Arrow);

pair polarpoint(real r, real phi){
  return (r*cos(phi),r*sin(phi));
}
label("$r=\varphi$",polarpoint(5pi/4,r(5pi/4)),3NE);
```



Obr. 3.16: Křivka v polárních souřadnicích

3.7.6 Osy

Pro vykreslení os Asymptote nabízí několik funkcí. Především jsou to `xaxis()` a `yaxis()`.

```
void xaxis(picture pic=currentpicture, Label L="",
           axis axis=YZero, real xmin=-infinity,
           real xmax=infinity, pen p=currentpen,
           ticks ticks=NoTicks, arrowbar arrow=None,
           bool above=false);
```

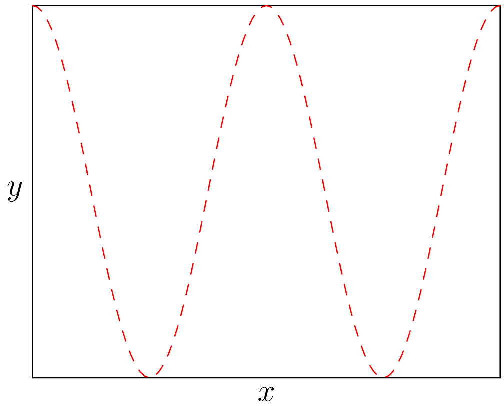
```
void yaxis(picture pic=currentpicture, Label L="",
           axis axis=XZero, real ymin=-infinity,
           real ymax=infinity, pen p=currentpen,
           ticks ticks=NoTicks, arrowbar arrow=None,
           bool above=false, bool autorotate=true);
```

Jak je z deklarací vidět, tyto funkce nemají žádné povinné argumenty. Pro základní osy bez popisků a šipek je tak možné zavolat pouze:

```
xaxis();
yaxis();
```

Popišme zde zbylé argumenty pro osu x (osa y analogicky). Specifikovat můžeme popisek `L`, rozsah vykreslení `xmin` a `xmax`. Dále můžeme nastavit styl šipky `arrow` (více o šipkách viz část 3.3), pero `p`, argument `above` určující zda se má osa vykreslit nad již existující objekty na plátně. Argument `axis` určuje umístění osy. Implicitně se osa x vykreslí na přímce $y = 0$. Pomocí tohoto argumentu je možné nechat vykreslit osu nahoře a/nebo dole na plátně. Slouží k tomu hodnoty `Bottom`, `Top` a `BottomTop` (pro osu y jsou to `Left`, `Right` a `LeftRight`). Běžné použití os je vidět na Obrázku 3.14. Zde si ukažme umístění os do krajů.

```
import graph;

size(0,150);

draw(graph(new real(real x){return 5*cos(x);},-2pi,2pi),
    dashed+red);

xaxis("$x$",axis=BottomTop);
yaxis("$y$",axis=LeftRight);
```



Obr. 3.17: Umístění os

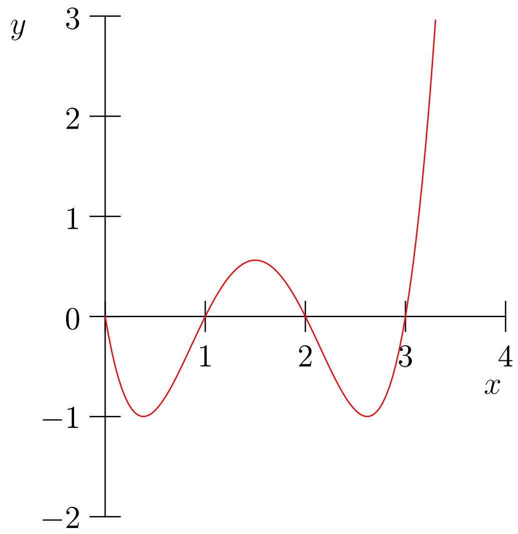
Posledním argumentem je `ticks`, tj. čárkování. Pro základní použití máme několik předdefinovaných hodnot: `LeftTicks`, `RightTicks` a `Ticks`. V takovém případě čárky budou nalevo, napravo a nebo obojí. Pro podrobnější možnosti využijeme stejnojmenné funkce `LeftTicks()`, `RightTicks()` a `Ticks()`. Význam argumentů je u všech analogický, popišme si proto jen první z jmenovaných.

```
ticks LeftTicks(Label format="", ticklabel ticklabel=null,
                bool beginlabel=true, bool endlabel=true,
                int N=0, int n=0, real Step=0, real step=0,
                bool begin=true, bool end=true,
                tickmodifier modify=None, real Size=0,
                real size=0, bool extend=false,
                pen pTick=nullpen, pen ptick=nullpen);
```

S `N` říkáme, na kolik intervalů se osa rozdělí velkými čárkami. S `n` je to podobné, udává počet intervalů mezi dvěmi velkými čárkami, které budou odděleny malými čárkami. Jiný způsob je specifikovat vzdálenosti `Step` příp. `step` mezi velkými příp. malými čárkami. Parametry `Size` a `size` určují velikost čárek.

```
import graph;

size(0,200);

real f(real x){
  return x*(x-1)*(x-2)*(x-3);
}

draw(graph(f,0,3.3),red);

xaxis("$x$",0,4,Ticks(beginlabel=false,n=1));
```

```
yaxis("$y$",-2,3,Ticks(n=1));
```



Obr. 3.18: Čárkování na osách

3.7.7 Popisky

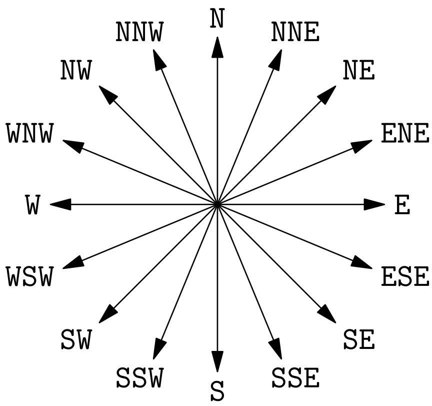
Pro vložení popisku na zadanou souřadnici můžeme použít funkci `label()`, pro vykreslení podél os pak funkce `labelx()` a `labely()`.

```
void label(picture pic=currentpicture, Label L,
           pair position, align align=NoAlign,
           pen p=nullpen, filltype filltype=NoFill);
```

```
void labelx(picture pic=currentpicture, Label L="",
            real x, align align=S, string format="",
            pen p=nullpen);
```

```
void labely(picture pic=currentpicture, Label L="",
            real y, align align=W, string format="",
            pen p=nullpen);
```

Rozeberme si nejprve funkci `label()`. Ta má jako povinné argumenty pouze popisek `L` (který můžeme zadat jako typ `string`) a souřadnice pro vykreslení `position`. Důležitý je také argument `align`, který určuje, v jakém směru a vzdálenosti od zadané pozice má být popisek umístěn. Většinou se používají konstanty odpovídající světovým stranám v angličtině (viz obr. 3.19), příp. jejich násobky.



Obr. 3.19: Přehled předdefinovaných směrů

Pro vykreslování popisků podél os využitím `labelx()`, příp. `labely()`, stačí zadat směr zarovnání a pozici na dané ose. Pokud například na ose y budeme chtít v bodě $y = 1$ vykreslit popisek 1, můžeme použít jednu z variant:

```
labely(1,E);
label("$1$",(0,1),E);
```

Pro další příklady viz ukázky v předchozích částech.

Kapitola 4

Práce ve 3D

V této části si uvedeme postupy pro generování 3D grafiky. Většinou se bude jednat o obdobu toho, co již bylo napsáno v předchozí části.

Popišme si stručně jak Asymptote pracuje při generování 3D grafiky pro PDF dokumenty. Nejprve se vygeneruje 3D model ve formátu PRC[1]. Pokud pracujeme v LaTeXu, vygeneruje se také náhledový obrázek. Ten se při další kompilaci LaTeXu automaticky vloží do dokumentu. Scéna se pak aktivuje až po kliknutí na tento obrázek. Někdy však můžeme chtít dokument bez interaktivního 3D modelu (kvůli velikosti, nebo protože tam není potřebný či je přímo nežádoucí). V takovém případě stačí přidat na začátek `asy` bloku pro daný obrázek příkaz:

```
settings.prc = false;
```

Jindy naopak můžeme chtít, aby se v dokumentu nacházel jen samotný model, tj. bez náhledového obrázku. V takovém případě stačí do kódu obrázku na začátek přidat:

```
settings.embed = false;
```

Ve 3D se také neobejdeme bez balíku `three`, který definuje zobecněné typy známé z 2D. Proto se téměř v celé této kapitole předpokládá jeho načtení:

```
import three;
```

[1]Specifikaci 3D formátu PRC lze nalézt na [8]

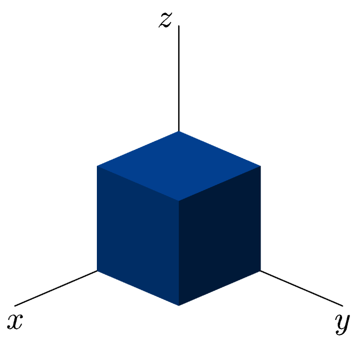
4.1 Kamera

Nastavení kamery se provádí pomocí promítání, které jsou typu `projection`. Promítání se mění pomocí proměnné `currentprojection`, kterou můžeme nastavit na rovnoběžné nebo perspektivní promítání.

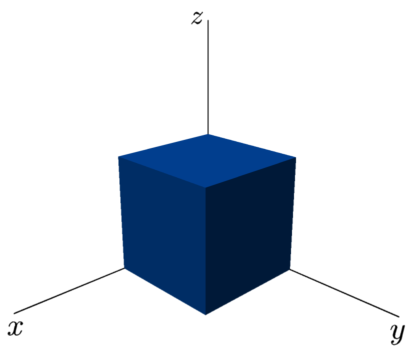
```
orthographic(triple camera, triple up=Z,
             triple target=O, real zoom=1,
             pair viewportshift=0, bool showtarget=true,
             bool center=false);
```

```
perspective(triple camera, triple up=Z,
            triple target=O, real zoom=1,
            real angle=0, pair viewportshift=0,
            bool showtarget=true, bool autoadjust=true,
            bool center=autoadjust);
```

Kamera bude umístěna do bodu (příp. ve směru bodu pro rovnoběžné promítání) `target+camera`, natočena směrem k `target`. Vektor `up` na obrazovce určuje směr nahoru (implicitně `(0,0,1)`). Obě tyto funkce existují ve variantě, kdy můžeme zadat jednotlivé složky vektoru `camera` zvlášť, výsledek je stejný. Pokud neuvedeme žádné promítání, použije se `perspective(5,4,2)`.



Obr. 4.1: `orthographic(3,3,2)`



Obr. 4.2: `perspective(3,3,2)`

Tím možnosti promítání nekončí. K dispozici je dále kosoúhlé promítání `obliqueX`, `obliqueY`, `obliqueZ`, případně si uživatel může definovat vlastní. Pro vyčerpávající popis viz [1, Index-oblique].

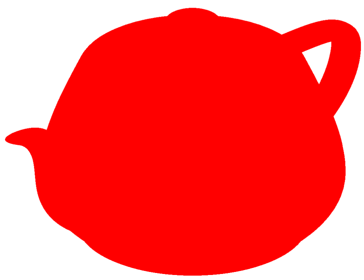
4.2 Světla

Nastavení světel je reprezentováno typem `light`, který je zadáván ve vykreslovacích příkazech. Pokud jej nezadáme, využije se `currentlight`, které je implicitně nastaveno na Headlamp. Další předdefinovaná světla jsou Viewport, White a pro scénu bez světel nolight (přesněji pro scénu bez stínů). Pro vlastní osvětlení můžeme využít funkci light:

```
light light(pen diffuse=white, pen ambient=black,
            pen specular=diffuse, pen background=nullpen,
            bool viewport=false, real x, real y, real z);
```

To umožňuje nastavit pera (především barvy) diffuse, ambient, specular a pero pro pozadí, background. Argument viewport říká, zda se má zdroj světla pohybovat společně s pohybem kamery. Zbylé argumenty určují souřadnice cíle světla. Pokud chceme ve scéně používat jedno nastavení světel, například nolight, stačí na začátku kódu uvést příkaz:

```
currentlight = nolight;
```



Obr. 4.3: nolight

Obr. 4.4: White

Obr. 4.5: Headlamp

Obr. 4.6: Viewport

4.3 Pera

Pro většinu kreslení ve 3D se využívá typ pen, známý ze 2D (viz část 3.2). Pro některé případy však balík three_light (importován balíkem three) zavádí rozšířený typ material. Ten uchovává především 4 pera/barvy diffuse, `ambient`, `emissive` a `specular` (uloženo v podobě per, umožňuje definovat i vzor). Dále uchovává (ne)průhlednost `opacity` a lesklost `shininess`. Materiál těchto vlastností vytvoříme pomocí:

```
material material(pen diffusepen=black, pen ambientpen=black,
                  pen emissivepen=black,
                  pen specularpen=mediumgray,
                  real opacity=opacity(diffusepen),
                  real shininess=defaultshininess);
```

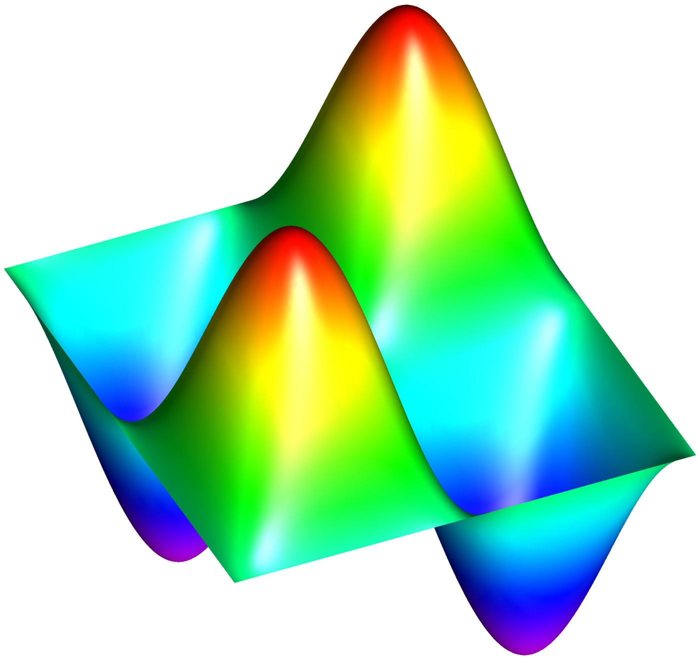
Většinou si vystačíme s materiálem indukovaným z pera typu `pen`. Takový materiál pak převezme zadané pero pro popis `diffuse` barvy a zbylé hodnoty doplní implicitními. Provedeme to například takto:

```
material m = black; //black je typu pen
```

Případně předáme typ `pen` přímo v místě, kde se zadává materiál, a převod se provede automaticky.

Zastavme se zde ještě u balíku `palette`. Ten nám umožňuje vykreslit oblasti či plochy se zadanou (nejen) barevnou paletou. Můžeme tak například zvýraznit výškové rozdíly v grafu, jak ilustruje následující ukázka. Více informací o tomto balíku viz [1, Documentation-8.28 palette].

```
import graph3;
import palette;

size(250,IgnoreAspect);

currentprojection=orthographic(2,4,1);
currentlight=Viewport;

real f(pair z) {
  real x=z.x,y=z.y;
  return sin(x)*sin(y);
}

surface s=surface(f,(0,0),(2pi,2pi),40,Spline);
s.colors(palette(s.map(zpart),Rainbow()));

draw(s);
```



Obr. 4.7: Využití balíku `palette`

4.4 Křivky

Stejně jako ve 2D, i ve 3D je základním kamenem všeho křivka. Balík `three` definuje typy `guide3` a `path3`, s kterými se pracuje téměř identicky jako s jejich 2D verzemi `guide` a `path`.

Stejně jako tomu bylo ve 2D, i zde můžeme při vykreslování dané křivky specifikovat typ šipky, případně příčky. Ty zadáváme při vykreslování pomocí `draw()`, nebo při používání nějaké vyšší funkce jako například `graph()` (viz 4.7). Použitelné hodnoty jsou `BeginArrow3`, `MidArrow3`, `EndArrow3`, `Arrow3` a `Arrows3` pro obyčejné šipky, dále `BeginArcArrow3`, `EndArcArrow3`, `ArcArrow3`, `MidArcArrow3` a `ArcArrows3` pro šipky s kratším hrotem. Pro příčky slouží `None`, `Blank`, `BeginBar3`, `EndBar3` a `Bar3`.

Více o křivkách vytvořených ze zadaného předpisu viz část 4.7.

4.5 Transformace

Transformace ve 3D se velice podobají těm ve 2D (viz 3.6). Jejich typ je `transform3`. Uveďme si zde jen v krátkosti seznam nejdůležitějších z nich.

```
transform3 shift(triple v);
```

```
transform3 xscale3(real x);
transform3 yscale3(real y);
transform3 zscale3(real z);
transform3 scale3(real s);
transform3 scale(real x, real y, real z);
```

```
transform3 rotate(real angle, triple v);
transform3 rotate(real angle, triple u, triple v);
```

```
transform3 reflect(triple u, triple v, triple w);
```

Použití lze vidět, mimo jiné, na obr.4.18 a 4.20.

4.6 Plochy

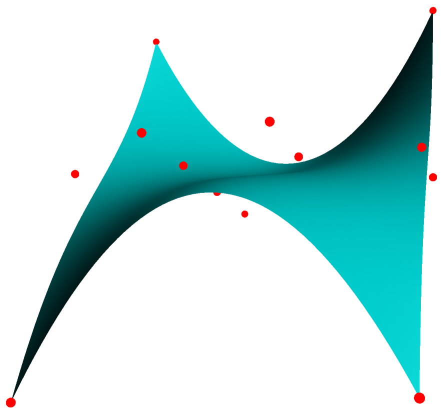
Plochy jsou v balíku **three** reprezentovány typem **surface**. Zde jsou uloženy jako pole Bézierových plátů (typ **patch**). Funkce v balících pro práci ve 3D často produkují typ **surface**, případně takový typ, který se na typ **surface** převádí, když je to potřeba (viz část 4.8). Následující funkce slouží pro vytvoření plochy výčtem kontrolních bodů.

```
surface surface(triple[][][] P,
                triple[][] normals=new triple[][],
                pen[][] colors=new pen[][],
                bool3 planar=default);
```

Tato varianta slouží pro případ, kdy chceme zadat plochu výčtem všech kontrolních bodů všech Bézierových plátů. Trojrozměrné pole **P** se očekává jako pole plátů, kde každý z plátů je očekáván jako pole čtyř čtveřic bodů. Dále lze pro rohové body nastavit normálové vektory **normals** a také barvy **colors**. Následuje ukázka vytvoření jednoho Bézierova plátu zadáním kontrolních bodů.

```
import three;

size(0,150);

currentprojection = perspective(5,2,-1);

triple[][][] P={
  {
    {(0.0, 0.0, 1.0),(0.5, 0.0, 0.0),
     (1.0, 0.0, 0.0),(1.5, 0.0, -1.0),},
    {(0.0, 0.5, 0.0),(0.5, 0.5, 0.0),
     (1.0, 0.5, 0.0),(1.5, 0.5,  0.0),},
```

```
    {(0.0, 1.0, 0.0),(0.5, 1.0, 0.0),
     (1.0, 1.0, 0.0),(1.5, 1.0,  0.0),},
    {(0.0, 1.5, 1.0),(0.5, 1.5, 0.0),
     (1.0, 1.5, 0.0),(1.5, 1.5, -1.0),}
  }
};

draw(surface(P),cyan);

for(int i=0;i<4;++i)
  for(int j=0;j<4;++j)
    dot(P[0][i][j],red);
```



Obr. 4.8: Bézierův plát zadán 16 kontrolními body

Dalšími způsoby pro vytváření ploch je generováním grafu funkce, vytažením ze zadaných křivek či rotací křivek. Více viz následující části.

4.7 Grafy funkcí

Při kreslení grafů funkcí v prostoru využijeme balík `graph3`[2]. jenž je obdobou balíku `graph` pro kreslení grafů funkcí v rovinně (viz 3.7).

4.7.1 Funkce $f(x, y)$

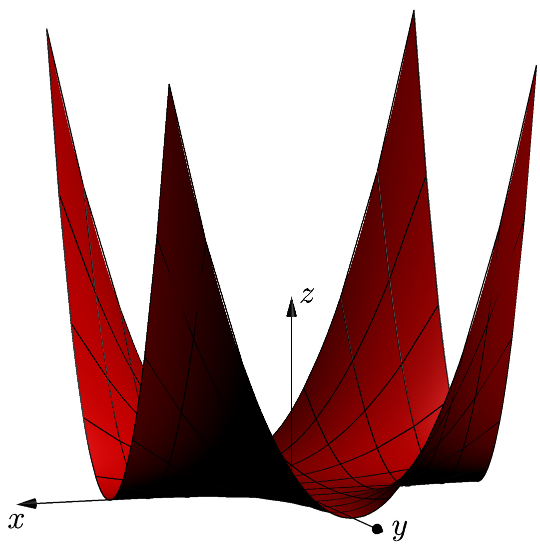
Graf funkce dvou proměnných vykreslíme tak, že pomocí funkce z balíku `graph3` vygenerujeme Bézierovu plochu typu `surface`, kterou posléze vykreslíme příkazem `draw`. Poslouží nám k tomu následující funkce.

[2]Balík `graph3` v sobě již zahrnuje načítání balíku `three` a ten tak není nutno načítat zvlášť.

```
surface surface(real f(pair z), pair a,
                pair b, int nx=nmesh, int ny=nx,
                splinetype xsplinetype,
                splinetype ysplinetype=xsplinetype,
                bool cond(pair z)=null);
```

Povinné argumenty jsou zde funkce dvou proměnných `f` a meze `a` (dolní meze) a `b` (horní meze). Celočíselné argumenty `nx` a `ny` určují dělení ve směru os x a y. Ukažme si použití na příkladu funkce $f(x, y) = |xy|^2$:

```
import graph3;

size(200,200,keepAspect=false);
currentprojection=orthographic(3,9,5);

real f(pair z){
    real x=z.x,y=z.y;
  return abs(x*y)^2;
}

draw(surface(f,(-2,-2),(2,2),xsplinetype=Spline),
red,meshpen=black+0.5);

xaxis3("$x$",-2,3,Arrow3);
yaxis3("$y$",-2,3,Arrow3);
zaxis3("$z$",0,7,Arrow3);
```

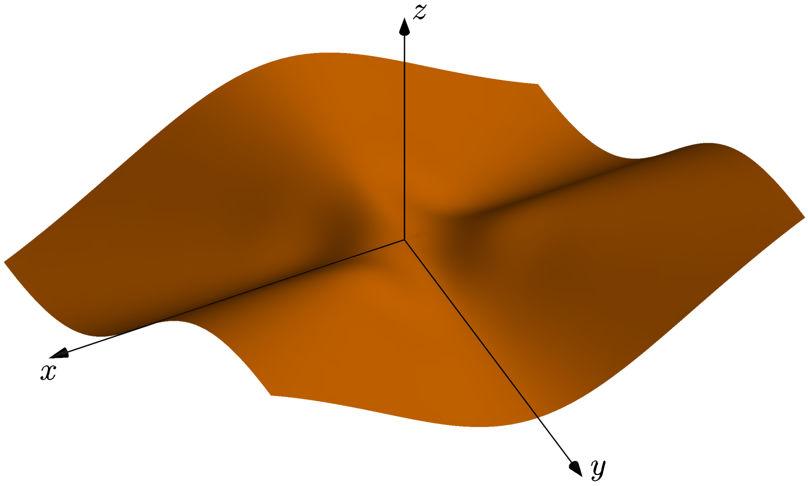


Obr. 4.9: Funkce $f(x, y) = |xy|^2$

4.7.2 Nespojité funkce

Pro nespojité funkce je v současné verzi Asymptote 2.14 pouze mizivá podpora a výsledky nevypadají příliš povedeně. Existuje sice stejný argument `cond` jako byl popsán v části 3.7.2, avšak graf je zpravidla velmi nekvalitní („zubatý“). Proto doporučuji použít vhodné parametrické vyjádření plochy či dodefinování funkce, rozdělení plochy na části apod. Následující příklad grafu funkce $f(x,y) = \frac{x^2 y}{x^2+y^2}$ ukazuje, jak lze funkci jednoduše dodefinovat v případě odstranitelné nespojitosti.

```
import graph3;

size(300);
currentprojection=orthographic(3,6,6);

real f(pair z){
  real x=z.x,y=z.y;
  if (x==0.0 && y==0.0) return 0;
  return (x^2*y)/(x^2+y^2);
}

draw(surface(f,(-3,-3),(3,3),xsplinetype=Spline),orange);

xaxis3("$x$",0,4,Arrow3);
yaxis3("$y$",0,4,Arrow3);
zaxis3("$z$",0,3,Arrow3);
```



Obr. 4.10: Funkce $f(x,y) = \frac{x^2 y}{x^2+y^2}$

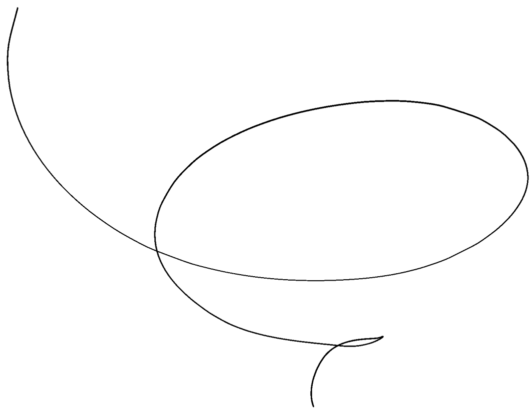
4.7.3 Parametrická křivka

Pro parametrickou křivku $f(t) = (x(t), y(t), z(t))$ máme následující variantu funkce `graph()`.

```
guide3 graph(picture pic=currentpicture, real x(real),
             real y(real), real z(real), real a, real b,
             int n=ngraph, interpolate3 join=operator --);
```

Předáváme tři funkce `x()`, `y()` a `z()`. Ty musí mít jeden argument typu `real` a vracet hodnotu typu `real` (tj. odpovídající reálné funkce jedné reálné proměnné). Dále zadáme meze intervalu `a` a `b`, případně dělení `n` a také typ interpolace `join`.

```
import graph3;

size(0,150);

triple f(real t){
  return (t*cos(t),t*sin(t),t);
}

draw(graph(f,0,4pi,join=operator ..,n=10));
```



Obr. 4.11: Prostorová křivka

4.7.4 Parametrická plocha

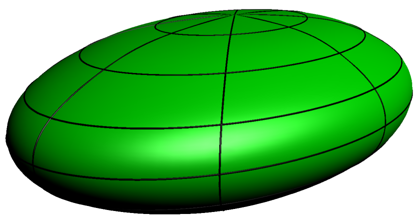
Plochu zadanou ve tvaru $F(u, v)$ můžeme vytvořit následující variantou funkce `surface()` z balíku `graph3`:

```
surface surface(triple f(pair z), pair a, pair b,
                int nu=nmesh, int nv=nu,
```

```
                    splinetype[] usplinetype,
                    splinetype[] vsplinetype=Spline,
                    bool cond(pair z)=null);
```

Zde dvojice hodnot `a` příp. `b` určuje dolní příp. horní meze pro oba parametry. Jemnost dělení lze nastavit pomocí `nu` a `nv`. Typ interpolace nastavíme pomocí `usplinetype`, příp. `vsplinetype`. U posledních zmíněných argumentů je dobré si všimnout, že jsou to pole typu `splinetype`, přesněji je očekáváno 3-prvkové pole. Hodnota `Spline` je však přetížena a lze ji zde použít (hodnoty `Straight`, `operator..` ani `operator--` však nelze). Pokud budeme chtít striktně lineární interpolaci, můžeme použít podobnou verzi `surface`, která se liší jen v tom, že nevyžaduje `usplinetype` ani `vsplinetype`.

```
import graph3;

size(150,0);

real a=3,b=2,c=1;
triple F(pair z){
  real phi=z.x, theta=z.y;
  return (a*cos(phi)*cos(theta),
          b*cos(phi)*sin(theta),
          c*sin(phi));
}

draw(surface(F,(-pi/2,0),(pi/2,2pi),nu=8,Spline),green,
   meshpen=black+0.5);
```



Obr. 4.12: Plocha zadaná parametricky

4.7.5 Implicitní funkce

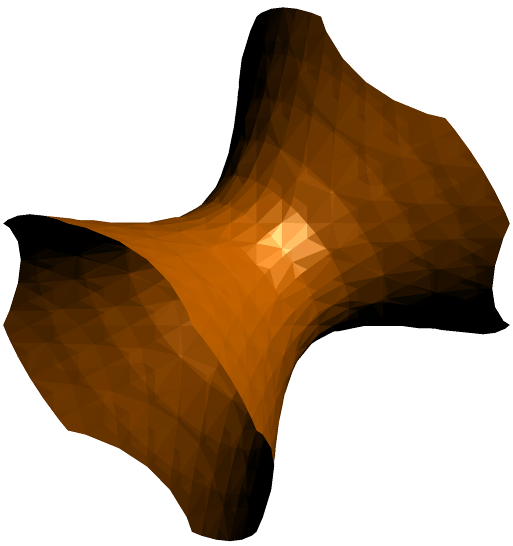
Plochu, zadanou v implicitním tvaru $F(x, y, z) = 0$, můžeme vykreslit funkcí `countour3` ze stejnojmenného balíku.

```
vertex[][] contour3(real F(real, real, real),
                    triple a, triple b,
                    int nx=nmesh, int ny=nx, int nz=nx,
```

```
                    projection P=currentprojection);
```

Body `a` a `b` jsou rohy příslušného kvádru, ze kterého se při sestrojování grafu volí hodnoty x, y a z.

```
import graph3;
import contour3;

size(0,200);

currentprojection = orthographic(1,1,2);

real F(real x, real y, real z){
  return -x^2+y^2+z^2-1;
}

draw(surface(contour3(F,(-2,-2,-2),(2,2,2),nx=10)),orange);
```



Obr. 4.13: Plocha zadaná implicitně

Pro lepší výsledky se však doporučuje použít parametrické vyjádření plochy.

4.7.6 Sférické a cylindrické souřadnice

Pro kreslení křivek zadaných ve sférických souřadnicích máme k dispozici funkci `polargraph` z balíku `graph3`. Očekávaný tvar křivky je

$$\theta = f(t),\ \varphi = g(t),\ r = h(\theta, \varphi),\ t \in [0, 1]$$

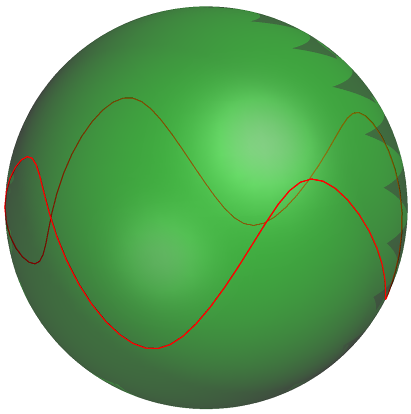
a příslušná funkce pro vytvoření

```
guide3 polargraph(real r(real,real), real theta(real),
                  real phi(real), int n=ngraph,
                  interpolate3 join=operator --);
```

Následuje příklad použití na křivce dané předpisem

$$\theta = \frac{1}{8}\sin(8\pi t)\pi + \frac{\pi}{2},\ \varphi = 2\pi t,\ r = 1,\ t \in [0,1].$$

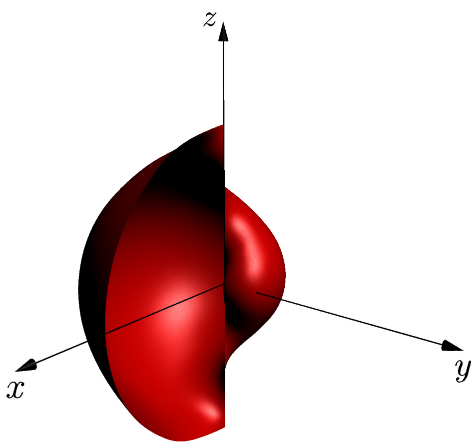
```
import solids;
import graph3;

size(0,150);

real phi(real t){return t*2pi;}
real theta(real t){return 1/8*sin(4*t*2pi)*pi+pi/2;}
real r(real theta, real phi){return 1;}

draw(surface(sphere(1)),green+opacity(0.5));
draw(polargraph(r,theta,phi),red);
```



Obr. 4.14: Křivka ve sférických souřadnicích

Balíky Asymptote v současné verzi nenabízí funkce pro vytváření křivek v cylindrických souřadnicích, ani pro plochy ve sférických či cylindrických souřadnicích. Není však těžké využít parametrického tvaru v kartézských souřadnicích, a pomocí nich vytvořit vlastní funkce. Následující dvě ukázky ilustrují takové dvě funkce `sphericalgraph` a `cylindricalgraph`. První ukázka představuje plochu danou předpisem $r(\theta,\varphi) = \frac{1}{5}\varphi$ ve sférických souřadnicích.

```
import graph3;

size(0,160);

surface sphericalgraph(real f(real, real), pair a, pair b,
 int nu=nmesh, int nv=nu, splinetype[] usplinetype=Spline,
 splinetype[] vsplinetype=Spline, bool cond(pair z)=null){
  triple F(pair z){
    real phi=z.x,theta=z.y;
    real r = f(phi,theta);
    return (r*cos(phi)*sin(theta),
            r*sin(phi)*sin(theta),
            r*cos(theta));
  }
  return surface(F,a,b,nu,nv,usplinetype,vsplinetype,cond);
}

real r(real phi,real theta){ return phi/5;}
draw(sphericalgraph(r,(0,0),(2pi,pi),nu=10),red);

xaxis3("$x$",0,2,Arrow3);
yaxis3("$y$",0,2,Arrow3);
zaxis3("$z$",0,2,Arrow3);
```

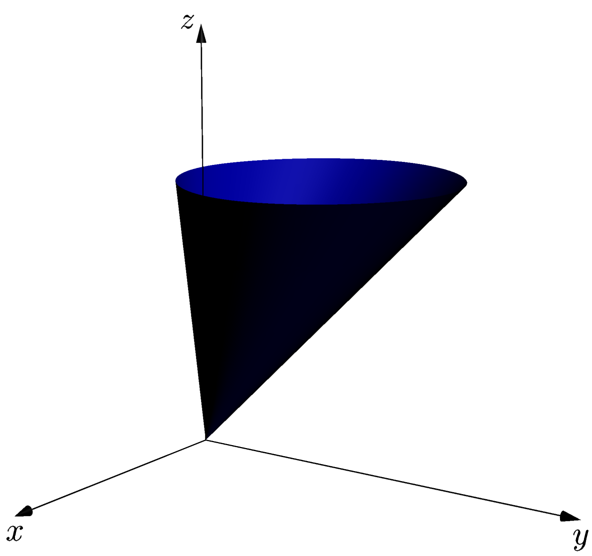


Obr. 4.15: Plocha ve sférických souřadnicích

Druhá ukázka představuje plochu danou předpisem $r(\varphi, z) = z\sin(\varphi)$ v cylindrických souřadnicích.

```
import graph3;

size(0,200);
```

```
surface cylindricalgraph(real f(real, real), pair a, pair b,
   int nu=nmesh, int nv=nu, splinetype[] usplinetype=Spline,
   splinetype[] vsplinetype=Spline, bool cond(pair z)=null){
  triple F(pair t){
    real phi=t.x,z=t.y;
    real r = f(phi,z);
    return (r*cos(phi),r*sin(phi),z);
  }
  return surface(F,a,b,nu,nv,usplinetype,vsplinetype,cond);
}

real r(real phi,real z){ return sin(phi)*z;}
draw(cylindricalgraph(r,(0,0),(2pi,1),nu=8,nv=4),blue);

xaxis3("$x$",0,1,Arrow3);
yaxis3("$y$",0,1.5,Arrow3);
zaxis3("$z$",0,1.5,Arrow3);
```



Obr. 4.16: Plocha v cylindrických souřadnicích

4.7.7 Osy

Pro vykreslení os ve 3D použijeme funkce `xaxis3()`, `yaxis3()` a `zaxis3()` z balíku `graph3()`. Formát jejich hlavičky je analogický, uveďme si jej pro `xaxis3()`.

```
void xaxis3(picture pic=currentpicture, Label L="",
```

```
              axis axis=YZZero, real xmin=-infinity,
              real xmax=infinity, pen p=currentpen,
              ticks3 ticks=NoTicks3, arrowbar3 arrow=None,
              bool above=false);
```

Funkce se podobá funkci `xaxis()` pro 2D případ, jen `axis`, `ticks` a `arrow` používají své trojrozměrné varianty. Šipka `arrow` může nabývat jedné z hodnot uvedených v části 4.4. Popišme si zbylé argumenty pro určení polohy os a čárkování.

Argument `axis` může nabývat hodnoty `YZZero` a potom osa x leží na přímce $y = 0$, $z = 0$ (jde o implicitní hodnotu). Analogicky, pro osy y a z máme hodnoty `XZZero` a `XYZero`. Pro specifikaci vlastního umístění můžeme použít funkce `YZEquals()`, `XZEquals()` nebo `XYEquals()`.

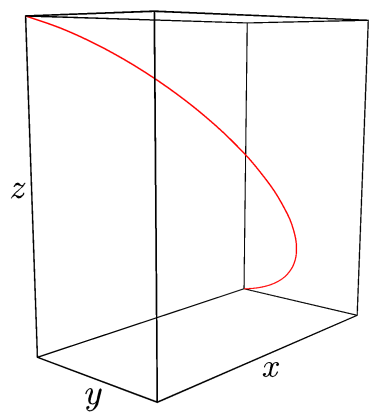
```
axis YZEquals(real y, real z);
axis XZEquals(real x, real z);
axis XYEquals(real x, real y);
```

Případné použití by pak mohlo vypadat třeba takto:

```
//osa x bude lezet na primce y=2, z=3
xaxis3("$x$",YZEquals(2,3));
```

Zbývajícím typem umístění je `Bounds`, které umístí osu do všech čtyř pozic podle rozměrů aktuálního modelu. Jedná se o trojrozměrnou analogii k `BottomTop` či `TopRight`.

```
import graph3;

size(0,150);

draw((0,0,0)..(1,1,1)..(2,0,2),red);

xaxis3("$x$",Bounds);
yaxis3("$y$",Bounds);
zaxis3("$z$",Bounds);
```



Obr. 4.17: Osy s využitím `Bounds`

Zbylý argument `ticks` určuje styl čárkování na osách. Přednastavené hodnoty jsou `NoTicks3`, `InTicks`, `OutTicks` a `InOutTicks`. Pro pokročilejší čárkování můžeme volat stejnojmenné funkce a specifikovat například počet čárek, krokování apod. Uveďme si hlavičku alespoň jedné z nich.

```
ticks3 InOutTicks(Label format="", ticklabel ticklabel=null,
                  bool beginlabel=true, bool endlabel=true,
                  int N=0, int n=0, real Step=0, real step=0,
                  bool begin=true, bool end=true,
                  tickmodifier modify=None, real Size=0,
                  real size=0, bool extend=false,
                  pen pTick=nullpen, pen ptick=nullpen);
```

Význam argumentů je zde totožný jako ve 2D, viz část 3.7.6. Důležité jsou zde především `N`, `n` (počet dílků oddělených velkými příp. malými čárkami), `Step`, `step` (vzdálenost mezi velkými příp. malými čárkami), `Size` a `size` (velikost velkých příp. malých čárek).

4.7.8 Popisky

Obdobně, jako tomu bylo ve 2D, také ve 3D můžeme sázet LaTeXové popisky. Využijeme k tomu:

```
void label(picture pic=currentpicture, Label L,
           triple position, align align=NoAlign,
           pen p=currentpen, light light=nolight,
           string name="", render render=defaultrender,
           interaction interaction =
           settings.autobillboard ? Billboard : Embedded);
```

Význam je analogický jako v odpovídající 2D variantě, popisek `L` se vysází na pozici `position`. Za povšimnutí stojí argument `interaction` určující, zda se má popisek natáčet s pohybem kamery (hodnota `Billboard`, implicitní chování), a nebo zda se má natočit staticky ve směru zadané projekce (hodnota `Embedded`).

Po načtení balíku `graph3` máme k dispozici i varianty pro snazší sázení popisků kolem os.

```
void labelx(picture pic=currentpicture, Label L="", triple v,
            align align=-Y, string format="", pen p=nullpen);
```

Analogicky `labely()` a `labelz()`.

4.8 Rotační tělesa

S využitím typu `revolution` z balíku `solids` můžeme vytvářet plochy vzniklé rotací křivky kolem zadaného vektoru. Typ `revolution` v sobě obsahuje právě tyto údaje, tj. křivku, která se otáčí a vektor, kolem kterého se otáčí. Musíme ho proto převést na typ `surface`, abychom získali plochu. Podívejme se, jak vytvářet proměnné typu `revolution`.

```
revolution cylinder(triple c=0, real r,
                    real h, triple axis=Z);
```

```
revolution cone(triple c=0, real r,
                real h, triple axis=Z,
                int n=nslice);
```

```
revolution sphere(triple c=0, real r, int n=nslice);
```

V případě válce (`cylinder`) zadáváme střed podstavy `c`, poloměr podstavy `r`, výšku `h` a směr válce `axis`. Podobně je to s kuželem (`cone`) a koulí (`sphere`), kde můžeme navíc zadat jemnost dělení stran `n`.

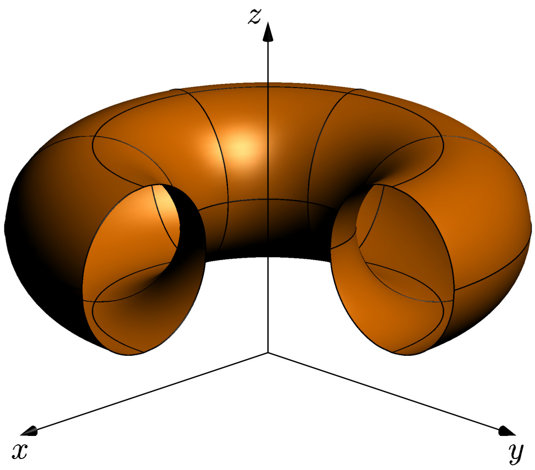
S těmito základními tvary bychom si však těžko vystačili. Ukažme si proto, jak vytvořit plochu vzniklou rotací obecné křivky kolem zadané osy.

```
revolution revolution(triple c=0, path3 g, triple axis=Z,
                      real angle1=0, real angle2=360);
```

Funkci předáme střed otáčení `c` (bod, z nějž povede osa otáčení), křivku `g`, osu otáčení `axis` (implicitně ve směru osy z, může však být libovolný vektor) a rozsah úhlů otáčení `angle1` a `angle2` ve stupních.

```
import solids;
```

```
size(0,175);
currentprojection=orthographic(2,2,1);

path3 p = rotate(90,(1,0,0))*shift(2,1.5,0)*unitcircle3;

revolution r=revolution(p,90,360);

draw(surface(r,n=5),orange,meshpen=black+0.5);

xaxis3("$x$",0,4,Arrow3);
yaxis3("$y$",0,4,Arrow3);
zaxis3("$z$",0,4,Arrow3);
```



Obr. 4.18: Plocha vzniklá rotací kružnice

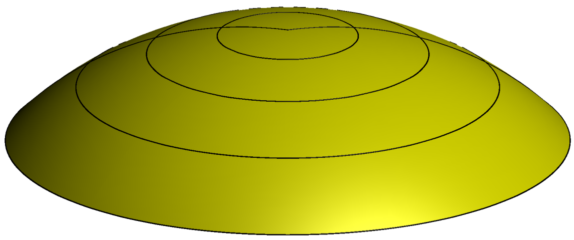
Ukažme si ještě, jak vytvořit plochu vzniklou rotací funkce $z = f(x)$.

```
revolution revolution(triple c=O, real f(real x),
                      real a, real b, int n=ngraph,
                      interpolate3 join=operator --,
                      triple axis=Z, real angle1=0,
                      real angle2=360);
```

Zde `c` je střed otáčení, `f` je příslušná funkce v mezích `a` až `b` s jemností dělení `n`. Dále je možné specifikovat typ interpolace argumentem `join`, osu otáčení `axis` a meze úhlu rotace `angle1` a `angle2`. Obr. 4.8 představuje plochu vzniklou rotací křivky, která je určena předpisem $z = \sqrt{1-x^2}$ pro $x \in [0, \sqrt{2}/2]$, kolem osy z.

```
import solids;
```

```
size(0,150);
currentprojection=orthographic(2,2,1);

real f(real x) {
  return sqrt(1-x^2);
}

revolution r=revolution(f,0,sqrt(2)/2,n=4,operator ..,Z);

draw(r,black+0.5);
draw(surface(r,n=5),yellow);
```



Obr. 4.19: Rotační plocha

4.9 Vytažené plochy

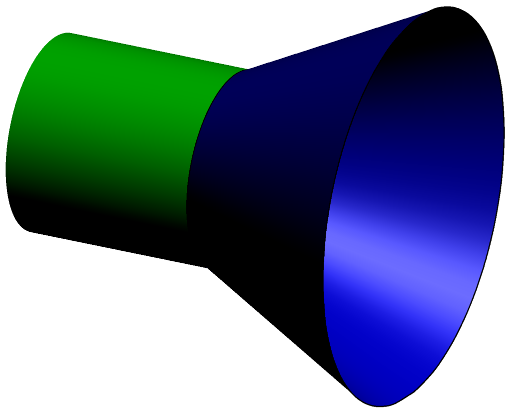
Dalším užitečným způsobem, jak vytvářet plochy, je vytažení vhodných křivek v určitém směru, případně mezi dvěmi křivkami. Balík `three` pro tento účel definuje několik verzí funkce `extrude()`. Uveďme si dvě základní varianty.

```
surface extrude(path3 p, triple axis=Z);
surface extrude(path3 p, path3 q);
```

V první verzi je křivka `p` vytažena ve směru `axis` (o délku vektoru `axis`). Druhá verze vytváří přímkovou plochu mezi dvěma zadanými křivkami. Pro ilustraci opět uvádíme příklad.

```
import three;

size(0,150);
```

```
currentprojection = orthographic(1,-2,1);

//dve kruznice
path3 p = rotate(90,(0,1,0))*unitcircle3;
path3 q = shift((2,0,0))*scale3(2)*p;

draw(p);
draw(q);

draw(extrude(p,(-2,0,0)),green);
draw(extrude(p,q),blue);
```

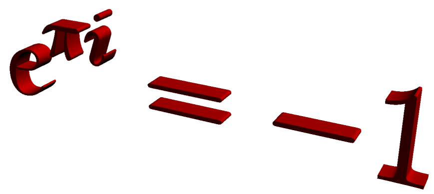


Obr. 4.20: Vytažené plochy

Kapitola 5

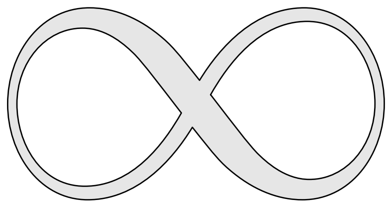
Algoritmy pro 3D reprezentaci rovinných oblastí

Jednou z vlastností Asymptote, odlišujících ho od podobných programů, je schopnost převádět rovinné systémy křivek (jakými je například font) do reprezentace plochami. S takovými objekty je pak možno pracovat jako s jinými 3D objekty, tj. aplikovat transformace, vytažení apod.



Obr. 5.1: 3D text vytažený z $e^{\pi i} = -1$

K tomu Asymptote používá algoritmus Bezulace (`BEZULATE`). Bezulace přebírá jednoduše souvislou oblast a postupem, podobným triangulaci, ji převádí na systém menších, jednodušších oblastí (Bézierových plátů). Prvotní oblast je určená systémem jednoduše uzavřených křivek. Těm však obecně neodpovídá jednoduše souvislá oblast (obr. 5.2). Z tohoto důvodu se před Bezulací provádí algoritmus `PARTITION`, který zadané oblasti rozdělí tak, že už budou jednoduše souvislé.



Obr. 5.2: Oblast určená znakem ∞

5.1 Rozdělení (`PARTITION`)

Předpokládejme konečnou množinu jednoduše uzavřených křivek. Těm odpovídají nějaké oblasti. Tento algoritmus je rozdělí na menší, jednoduše souvislé oblasti (ve tvaru množiny křivek). Provede se to tak, že se křivky nejdříve seřadí podle toho, jak jsou v sobě vzájemně obsaženy. Takové rekurzivní seřazení provádí procedura `SORT`. Pak se mezi vnějšími a vnitřními křivkami vytváří vhodné úsečky, které dané oblasti propojují (procedura `MERGE`).

Algoritmus 1 `PARTITION(D)`

Vstup: množina jednoduše uzavřených křivek `D`
Výstup: množina uzavřených křivek `A`

```
A ← ∅
for G ∈ SORT(D) do
  innerGroups ← SORT(G.inners)
  for H ∈ innerGroups do
    A ← A ∪ PARTITION(H.inners)
  innerTops ← ∅
  for K ∈ innerGroups do
    innerTops ← innerTops ∪{K.top}
  A ← A ∪ MERGE(G.top,innerTops)
return A
```

Popišme si použité procedury `SORT` a `MERGE`. Jak již bylo zmíněno, procedura `SORT` seřadí křivky v závislosti na tom, jak jsou v sobě obsaženy. To se dá provést efektivně, protože křivky jsou jednoduše uzavřené. Na test pak stačí použít libovolný bod křivky.

Algoritmus 2 SORT(D)

Vstup: množina jednoduše uzavřených křivek D

Výstup: množina A dvojic „vrchních“ a příslušných, v nich obsažených křivek

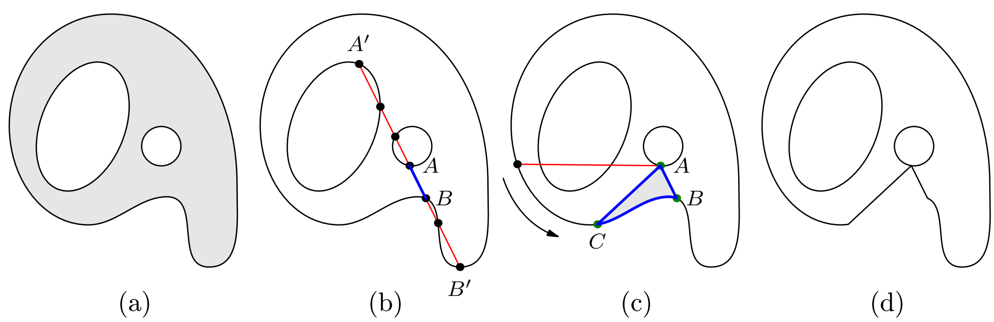
```
G ← ∅
for C ∈ D do
  found ← false
  for H ∈ G do
    if C leží uvnitř H.top then
      H.inners ← H.inners ∪ {C}
      found ← true
      break
    else if H.top leží uvnitř C then
      H.inners ← H.inners ∪ {H.top}
      H.top ← C
      for H' ∈ G \ {H} do
        if H'.top leží uvnitř C then
          H.inners ← H.inners ∪ H'.inners
          H.inners ← H.inners ∪ {H'.top}
          G ← G \ {H'}
      found ← true
      break
  if not found then
    G ← G ∪{(C,∅)}
return G
```

Procedura MERGE prochází křivky inners, které jsou obsaženy v křivce top, a snaží se je vhodnými úsečkami propojit (obr. 5.3). Libovolný bod na vnější křivce spojí s libovolným bodem na vnitřní křivce. Výsledná přímka pak obsahuje dva průsečíky, A s vnitřní, B s vnější křivkou takové, že mezi A a B se již jiné průsečíky nenachází (b). Dále se hledá bod C na vnější křivce takový, že přímka AC, kromě krajních bodů, neprotíná žádnou z křivek a zároveň oblast určená úsečkami $\overline{AB}$, $\overline{AC}$ a křivkou $\overline{BC}$ (úsek na vnější křivce) neobsahuje žádnou další z křivek v inners (c). To se nám po případném opakovaném dělení segmentů sousedících s B vždy podaří. Nakonec se nalezená oblast oddělí a pokračuje se se zbylými křivkami (d).

Algoritmus 3 `MERGE`(`top`,`inners`)

Vstup: křivka `top` a množina křivek `inners` obsažených v `top`
Výstup: množina `A` uzavřených křivek

`A` ← ∅
while `inners` ≠ ∅ **do**
    `D` ← libovolná křivka z `inners`
    $\overline{\texttt{A'B'}}$ ← úsečka mezi libovolnými body `B'` na `top` a `A'` na `D`
    `B` ← ten z průsečíků $\overline{\texttt{A'B'}}$ a `top` nejblíž bodu `A'`
    `t` ← čas bodu `B` na křivce `top`
    `A` ← ten z průsečíků $\overline{\texttt{BA'}}$ s křivkami z `inners` nejblíže bodu `B`
    `D'` ← křivka z `inners`, na které se realizuje průsečík `A`
    $\Delta \leftarrow 2$
    `found` ← **false**
    repeat
        $\Delta \leftarrow \Delta/2$
        for `sgn` $\in \{-1, 1\}$ **do**
            `C` ← bod na křivce `top` v čase `t`+`sgn`Δ
            `l` ← subpath(`top`,`B`,`C`) + $\overline{\texttt{CA}}$ + $\overline{\texttt{AB}}$
            if `l` neobsahuje ani neprotíná žádnou křivku z `inners` s výjimkou bodu `A` **then**
                `found` ← **true**
    until `found`
    `A` ← `A` ∪ {`l`}
    `top` ← subpath(`top`,`C`,`B`) + $\overline{\texttt{BA}}$+ subpath(`D'`,`A`,`A`) +$\overline{\texttt{AC}}$
    `inners` ← `inners` \ {`D'`}
`A` ← `A` ∪ {`top`}
return `A`



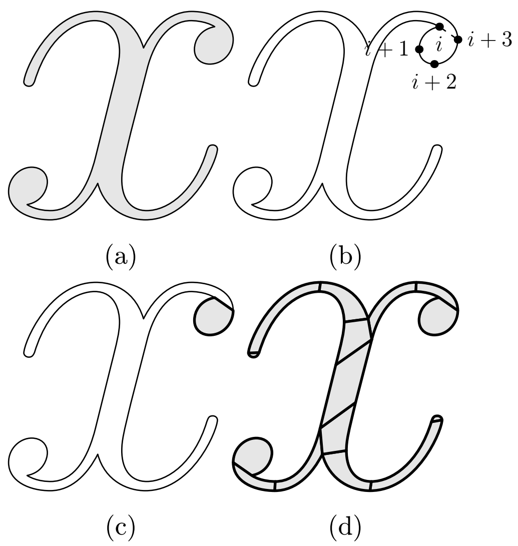
Obr. 5.3: procedura `MERGE`

Ve výsledku se tedy `PARTITION` redukuje na zjištění, zda bod leží uvnitř křivky a na počítání průsečíků přímky a křivky. Počítání průsečíků Bézierovy kubiky s přímkou vede na hledání kořenů kubického polynomu, které dovedeme spočítat. Pro zjištění, zda bod leží uvnitř oblasti zadané uzavřenou křivkou, se může použít např. Winding number (způsob použitý v Asymptote).

5.2 Bezulace

Dostáváme se k popisu algoritmu Bezulace, hlavního algoritmu této části. Na vstupu již předpokládáme jednoduše uzavřenou křivku C, jíž odpovídá jednoduše souvislá oblast. Účelem Bezulace je tuto oblast převést na sjednocení menších oblastí ohraničených nejvýše čtyřmi Bézierovými kubikami. Popišme, jakým způsobem Bezulace sestavuje tyto ohraničující křivky.

Nechť křivka C prochází n kontrolními body $P_0, P_1, \ldots, P_{n-1}$ (body mezi jednotlivými segmenty). Algoritmus postupně prochází body P_i a testuje, zda oblast, určená segmenty P_iP_{i+1}, $P_{i+1}P_{i+2}$, $P_{i+2}P_{i+3}$ a úsečkou $\overline{P_{i+3}P_i}$, leží uvnitř křivky C. Na to stačí zkontrolovat jen zmíněnou úsečku. V případě, že tam oblast leží, vhodnou rekonstrukcí křivky se tato oblast „vyjme“, a algoritmus se opakuje pro zbylou část. Pokud se vhodnou oblast, tj. trojici případně čtveřici bodů, nepodaří nalézt, křivka C se zjemní přidáním dodatečného bodu doprostřed každého segmentu. Celý proces se pak opakuje, dokud délka původní křivky neklesne na 4 a méně. Při dostatečně jemném dělení se původní křivka stává nerozlišitelnou od mnohoúhelníku a celý proces se redukuje na triangulaci. Algoritmus se proto vždy dokončí.



Obr. 5.4: Algoritmus Bezulace

Algoritmus 4 BEZULATE(D)

Vstup: množina uzavřených křivek D
Výstup: množina uzavřených křivek A délek 3 nebo 4

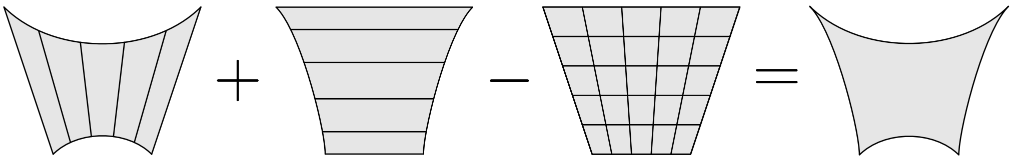
```
A ← ∅
for C ∈ D do
  while C.length > 4 do
    found ← false
    for n = 3 to 2 do
      for i = 0 to C.length-1 do
        L ← úsečka mezi uzly i a i+n na C
        if L protíná C právě ve dvou bodech and
        prostřední bod L leží uvnitř C then
          p ← subpath(C,i,i + n)
          q ← subpath(C,i + n,i + C.length)
          A ← A ∪ {p + L}
          C ← L + q
          found ← true
          break
      if found then
        break
    if not found then
      zjemni C přidáním bodu doprostřed každého segmentu
return A
```

5.3 Coonsova bilineární plocha

Posledním krokem je získání Bézierova plátu ze čtyř okrajových Bézierových křivek. K tomu využijeme obecný postup, tzv. Coonsovu bilineární plochu, která je určená vztahem:

$$
\begin{aligned}
Q(u,v) = (1-u,u)\begin{pmatrix} Q(0,v) \\ Q(1,v) \end{pmatrix} + (Q(u,0),Q(u,1))\begin{pmatrix} 1-v \\ v \end{pmatrix} - \\
-(1-u,u)\begin{pmatrix} Q_{0,0} & Q_{0,1} \\ Q_{1,0} & Q_{1,1} \end{pmatrix}\begin{pmatrix} 1-v \\ v \end{pmatrix}
\end{aligned}
\tag{5.1}
$$

kde $u,v \in [0,1] \times [0,1]$. Body $Q(0,v)$, $Q(1,v)$, $Q(u,0)$ a $Q(u,1)$ známe, poněvadž leží na okrajových křivkách.



Obr. 5.5: Coonsova bilineární plocha

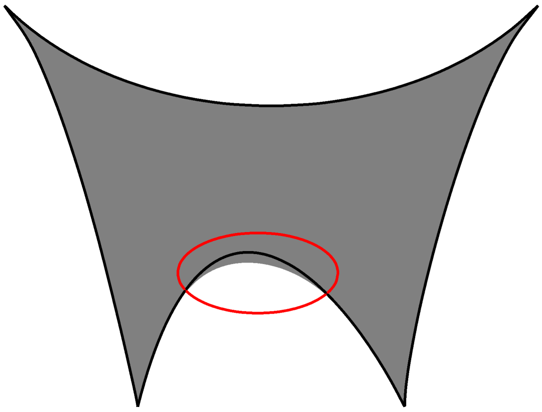
V našem případě však chceme plochu v podobě Bézierova plátu. Proto potřebujeme z předchozího obecného zápisu vyjádřit 4 vnitřní body pro speciální případ s Bézierovými okrajovými křivkami. Bézierův plát je zadán 16 body $P_{i,j}$ a předpisem:

$$f(u,v) = \sum_{i,j=0}^{3} B_i(u)B_j(v)P_{i,j} \quad \text{pro} \quad u,v \in [0,1] \tag{5.2}$$

kde $B_i(t) = \binom{3}{i} t^i (1-t)^{3-i}$ je Bernsteinův polynom třetího stupně. Známe všechny body $P_{i,j}$ kromě vnitřních $P_{1,2}$, $P_{2,1}$, $P_{1,1}$ a $P_{2,2}$. Položením rovnosti s výrazem 5.1 a porovnáním koeficientů dostaneme příslušné vztahy.

$$P_{1,2} = \frac{1}{9}(6P_{0,2} + 6P_{1,3} - 4P_{0,3} + 3P_{1,0} + 3P_{3,2} - 2P_{0,0} - 2P_{3,3} - P_{3,0})$$
$$P_{2,1} = \frac{1}{9}(6P_{2,0} + 6P_{3,1} - 4P_{3,0} + 3P_{0,1} + 3P_{2,3} - 2P_{0,0} - 2P_{3,3} - P_{0,3})$$
$$P_{1,1} = \frac{1}{9}(6P_{0,1} + 6P_{1,0} - 4P_{0,0} + 3P_{1,3} + 3P_{3,1} - 2P_{3,0} - 2P_{0,3} - P_{3,3})$$
$$P_{2,2} = \frac{1}{9}(6P_{3,2} + 6P_{2,3} - 4P_{3,3} + 3P_{2,0} + 3P_{0,2} - 2P_{3,0} - 2P_{0,3} - P_{0,0})$$

Tím celý algoritmus ještě nekončí. V případě, kdy by byl výsledný Bézierův plát degenerovaný, tj. nebyl by difeomorfizmem ze čtverce $[0,1] \times [0,1]$, vznikaly by při renderování grafické artefakty (obr. 5.6). Z tohoto důvodu se v Asymptote kontroluje postačující podmínka pro degenerovanost. Podmínka je založena na faktu, že Jakobián pro zmíněnou plochu musí být všude nenulový (více viz [2]).



Obr. 5.6: Grafický artefakt degenerovaných ploch

Výrazy

$$\sum_{i+k=p} \sum_{j+l=q} (P_{i+1,j} - P_{i,j}) \times (P_{k,l+1} - P_{k,l}) \binom{2}{i} \binom{3}{k} \binom{3}{j} \binom{2}{l} \tag{5.3}$$

pro $p, q = 0, 1, \ldots, 5$, musí mít stejné zmaménko, pokud je plocha nedegenerovaná. Zde operace $\times$ na vektorech z $\mathbb{R}^2$ odpovídá třetí složce vektorového součinu příslušných vektorů v $\mathbb{R}^3$, tj. $u \times v = u_x v_y - u_y v_x$. V případě, že se podaří degenerovanost zjistit, zkouší se dále degenerovanost na okraji plochy. K tomu slouží následující věta.

Věta 1. *Nechť p je uzavřená, kladně orientovaná, rovinná křivka délky 4. Dále nechť vnitřní úhly svírané vcházející a vycházející tečnou jsou ve všech jejích uzlech menší nebo rovny $180°$. Dále nechť $J(u, v)$ je Jakobián odpovídajícího Bézierova plátu s kontrolními body $P_{i,j}$. Položme*

$$f(u) = \frac{1}{3} J(u, 0) = \sum_{i,j=0}^{3} B'_i(u) B_j(u) P_{i,0} \times (P_{j,1} - P_{j,0}). \tag{5.4}$$

Jestliže pro všechna $u \in (0, 1)$ taková, že $f'(u) = 0$, platí $f(u) \geq 0$, pak $J(u, 0) \geq 0$ na $[0, 1]$. V opačném případě se minimální hodnota $J(u, 0)$ nachází v některém u takovém, že $f'(u) = 0$.

Důkaz. Předpokládejme, že pro všechna $u \in (0, 1)$, $f'(u) = 0$ platí $f(u) \geq 0$. Dále předpokládejme, že $J(u, 0) < 0$ pro nějaké $u \in [0, 1]$. Jelikož $J(u, 0)$ je spojitá funkce, nabývá svého minima v nějakém $\overline{u} \in [0, 1]$. Jistě je pak $J(\overline{u}, 0) < 0$. Dále platí

$$J(0, 0) = 3f(0) = 9(P_{1,0} - P_{0,0}) \times (P_{0,1} - P_{0,0}) \geq 0$$
$$J(1, 0) = 3f(1) = 9(P_{3,0} - P_{2,0}) \times (P_{3,1} - P_{3,0}) \geq 0$$

poněvadž se jedná o vektorové součiny tečných vektorů na křivce p, která je kladně orientovaná a úhly svírané těmito tečnami nepřesahují 180°. Proto $\overline{u} \in (0, 1)$. Protože však $J(u, 0) = 3f(u)$, nabývá svého minima v $\overline{u}$ i funkce f. Potom zřejmě $f'(\overline{u}) = 0$ a $f(\overline{u}) < 0$, což je spor. $\square$

Nejdřív tedy rozdělíme okraj v uzlech, kde úhel mezi vcházející a vycházející tečnou přesahuje 180°. Potom se numerickými metodami naleznou kořeny polynomu 4. stupně $f'(u)$ na $(0, 1)$. V nalezených bodech u se potom zkontrolují znaménka $f(u)$. V případě zápornosti již pak stačí spočítat $J(u, 0)$ ve všech nalezených kořenech, a tak zjistit, kde je Jakobián nejmenší. Takto spočítané body určují optimální rozdělení okrajových křivek. Daným způsobem se počet výsledných plátů snižuje oproti jiným, rychlejším metodám. To je v našem konkrétním případě výhodné, protože chceme vytvořit pouze reprezentaci původní oblasti a časová složitost je tu méně podstatná. Uvedenou Větu použijeme analogicky pro nalezení degenerace i na zbylých třech okrajových křivkách.

V případě, že se degenerace na okraji nezjistí, musí být degenerace uvnitř plochy. V takovém případě se rozdělení provede náhodně ve zvolených místech na okraji. Celý proces se potom rekurzivně opakuje.

Příloha

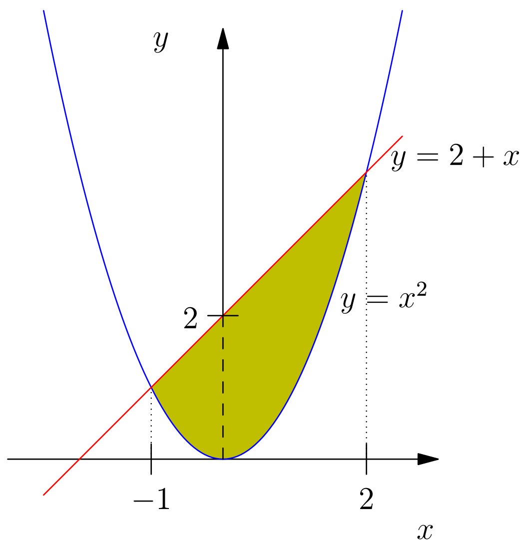
Zde se nachází grafika ztvárňující vybrané obrázky z [5]. Uváděná čísla obrázků proto odkazují do zmíněné literatury. U 3D modelů je zároveň použit příkaz

```
shipout(options="3Drender=SolidOutline");
```

z důvodu optimalizace velikosti výsledného modelu. Černá síť potom není součástí modelu, ale je až vlastností zobrazení scény v Adobe Readeru. Tím se docílí menší velikosti souboru a rychlejšího nahrávání modelů, než při využití např. `meshpen=black+0.5`. Na uvedených obrázcích bylo tímto způsobem ušetřeno v průměru přes 200 kB na obrázek. To vše však za cenu absence sítě na náhledovém obrázku.

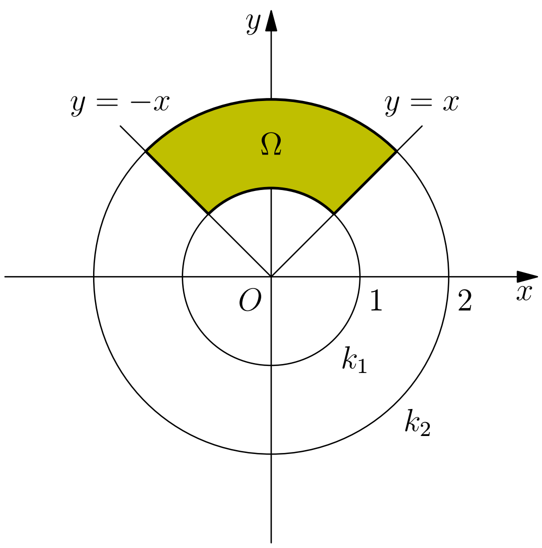
Uvedené modely by šly vytvořit i jinými způsoby, použité parametrizace ploch byly zvoleny především kvůli tvaru výsledné sítě.

```
import graph;

size(200);

real f(real x){return x^2;}
real g(real x){return 2+x;}

fill(graph(f,-1,2)--cycle,lightolive);
draw(graph(f,-2.5,2.5),blue);
draw(graph(g,-2.5,2.5),red);

draw((-1,0)--(-1,g(-1)),dotted);
draw((2,0)--(2,g(2)),dotted);

draw((0,0)--(0,2),dashed);
yaxis("$y$",2,6,Ticks(new real[]{2}),Arrow,above=true);
xaxis("$x$",-3,3,Ticks(new real[]{-1,2}),Arrow);

label("$y=2+x$",(2.2,g(2.2)),E);
label("$y=x^2$",(1.5,f(1.5)),E);
```



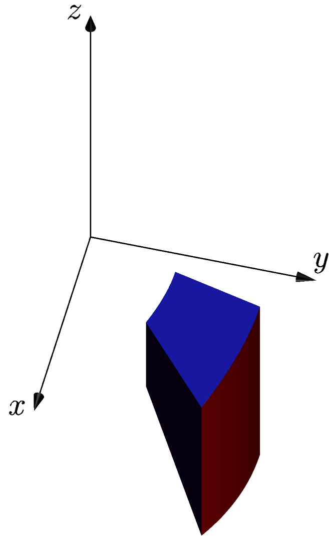
Obr. P.1: Obrázek 1.3

```
import graph;

size(200);

draw(unitcircle);
draw(scale(2)*unitcircle);
draw((0,0)--(1.7,1.7));
draw((0,0)--(-1.7,1.7));

path p = arc((0,0),2,45,135)--(-1,1)--arc((0,0),1,135,45)
          --cycle;
filldraw(p,lightolive,drawpen=black+1.0);

yaxis("$y$",-3,3,Arrow);
xaxis("$x$",-3,3,Arrow);

labelx("$O$",0,SW);
labelx("$1$",1,SE);
labelx("$2$",2,SE);

real sqrt2 = sqrt(2);
label("$\Omega$",(0,1.5),NoAlign);
label("$y=x$",(1.7,1.7),N);
label("$y=-x$",(-1.7,1.7),N);
label("$k_1$",(sqrt2/2,-sqrt2/2),SE);
label("$k_2$",(sqrt2,-sqrt2),SE);
```



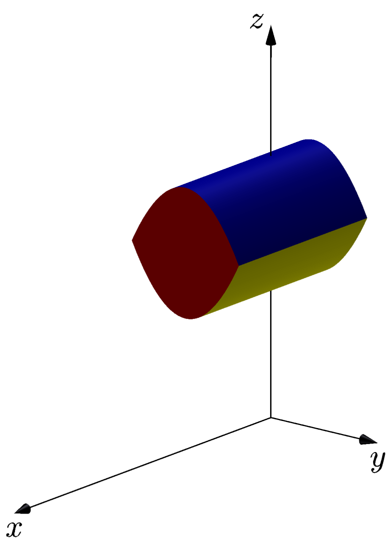
Obr. P.2: Obrázek 1.5

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(6,3,6,target=(2,2,1));

triple f1(real x){return (x,sqrt(2*x-x^2),sqrt(2*x-x^2));}
triple f2(real x){return (x,sqrt(4*x-x^2),sqrt(4*x-x^2));}

path3 p1 = graph(f1,1,3/2,n=8,operator..);
path3 p2 = (3/2,sqrt(3)/2,sqrt(3)/2)--(3,sqrt(3),sqrt(3));
path3 p3 = graph(f2,3,2,n=8,operator..);
path3 p4 = (2,2,2)--(1,1,1);
path3 q1 = path3(path(p1)), q2 = path3(path(p2)),
      q3 = path3(path(p3)), q4 = path3(path(p4));

draw(extrude(p1,q1),red);
draw(extrude(p2,q2),purple);
draw(extrude(p3,q3),red);
draw(extrude(p4,q4),purple);
draw(extrude(p1,reverse(p3)),blue);
draw(extrude(q1,reverse(q3)),yellow);

axes3("$x$","$y$","$z$",(0,0,0),(2,2,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



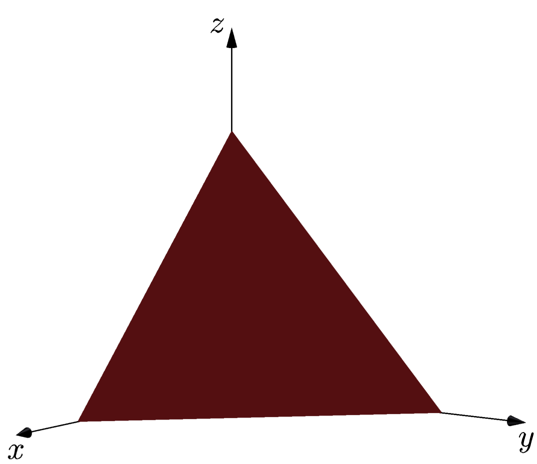
Obr. P.3: Obrázek 1.10

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

triple F1(real y){return (0,y,4-y^2);}
triple F2(real y){return (0,y,y^2+2);}

path3 p1 = shift(-1,0,0)*graph(F1,-1,1,n=10,operator..);
path3 p2 = shift(-1,0,0)*graph(F2,-1,1,n=10,operator..);

draw(extrude(p1,(3,0,0)),blue);
draw(extrude(p2,(3,0,0)),yellow);
draw(extrude(p1,p2),red);
draw(shift(3,0,0)*extrude(p1,p2),red);

axes3("$x$","$y$","$z$",(0,0,0),(6,2,6),Arrow3);

shipout(options="3Drender=SolidOutline");
```



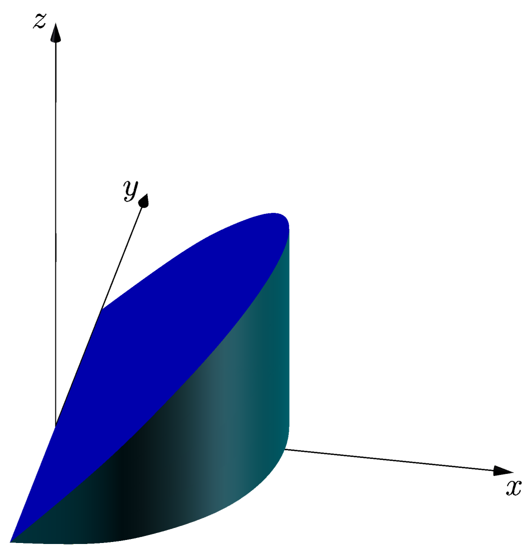
Obr. P.4: Obrázek 2.3

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2,target=(5/4,5/4,5/4));

triple x=(5,0,0),y=(0,5,0),z=(0,0,5);

draw(surface(x--y--z--cycle),red);
draw(surface(x--z--O--cycle),cyan);
draw(surface(y--z--O--cycle),blue);
draw(surface(x--y--O--cycle),yellow);

axes3("$x$","$y$","$z$",(0,0,0),(7,7,7),Arrow3);

shipout(options="3Drender=SolidOutline");
```



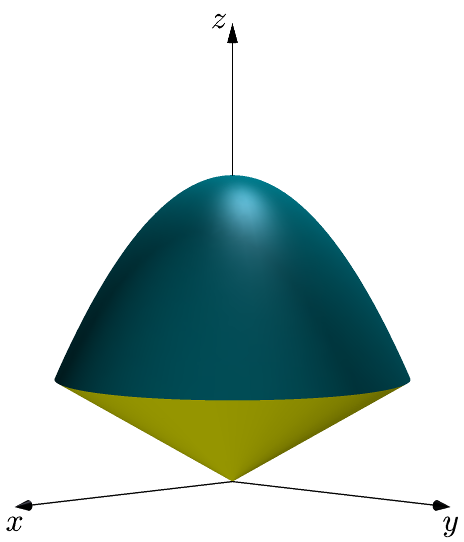
Obr. P.5: Obrázek na str. 68

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(1,-5,3);

triple upper(pair z){
  return (z.x*cos(z.y),z.x*sin(z.y),z.x*cos(z.y));
}
triple bottom(pair z){
  return (z.x*cos(z.y),z.x*sin(z.y),0);
}
triple back(pair z){
  return (cos(z.y),sin(z.y),z.x*cos(z.y));
}

draw(surface(upper,(0,-pi/2),(1,pi/2),nu=4,nv=10,Spline),
      blue);
draw(surface(bottom,(0,-pi/2),(1,pi/2),nu=4,nv=10,Spline),
      yellow);
draw(surface(back,(0,-pi/2),(1,pi/2),nu=4,nv=10,Spline),
      cyan);

axes3("$x$","$y$","$z$",(0,0,0),(2,2,2),Arrow3);

shipout(options="3Drender=SolidOutline");
```



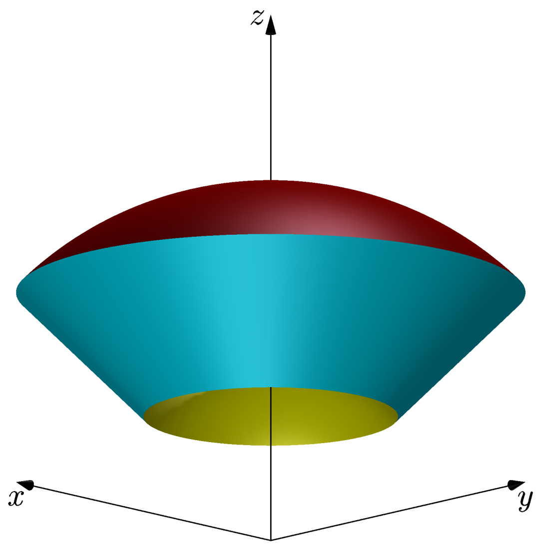
Obr. P.6: Obrázek 2.7

```
import graph;
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(6,6,1);

real xmax = 2/sqrt(3);
real f1(real x){return 2-x^2;}
real f2(real x){return x/sqrt(3);}

draw(surface(revolution(f1,0,xmax,n=10,operator..,Z)),cyan);
draw(surface(revolution(f2,0,xmax,n=4,Z)),yellow);

axes3("$x$","$y$","$z$",(0,0,0),(2,2,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



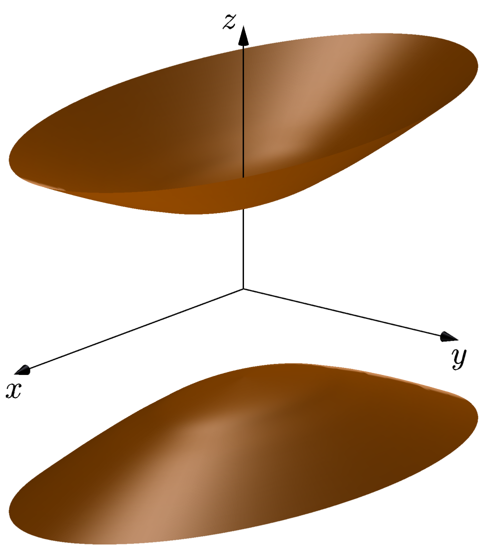
Obr. P.7: Obrázek 2.11

```
import graph;
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(6,6,-2);

real xmax1 = 1/sqrt(2);
real xmax2 = sqrt(2);
real f1(real x){return sqrt(1-x^2);}
real f2(real x){return x;}
real f3(real x){return sqrt(4-x^2);}

draw(surface(revolution(f1,0,xmax1,n=8,operator..,Z)),
     yellow);
draw(surface(revolution(f2,xmax1,xmax2,n=4,Z)),cyan);
draw(surface(revolution(f3,0,xmax2,n=10,operator..,Z)),red);

axes3("$x$","$y$","$z$",(0,0,0),(2,2,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



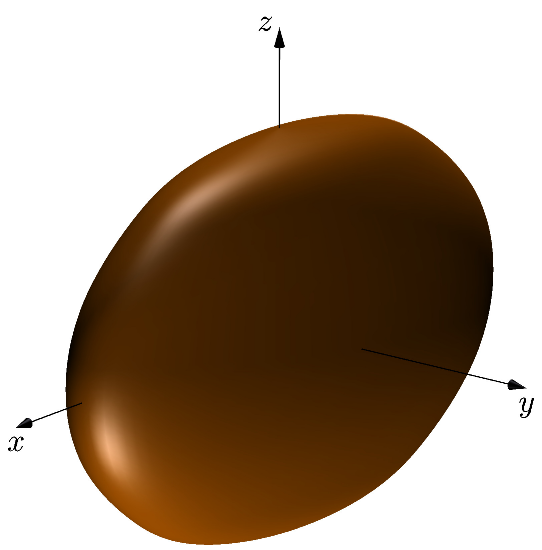
Obr. P.8: Obrázek 2.15

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=2,b=1,c=1;
triple f(pair z){
  real u=z.x, v=z.y;
  return (a*sqrt(-1+u^2)*cos(v),b*sqrt(-1+u^2)*sin(v),c*u);
}

surface upper = surface(f,(1,0),(2,2pi),Spline,nu=8);

draw(upper,orange);
draw(rotate(180,(0,1,0))*upper,orange);

axes3("$x$","$y$","$z$",(0,0,0),(4,3,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



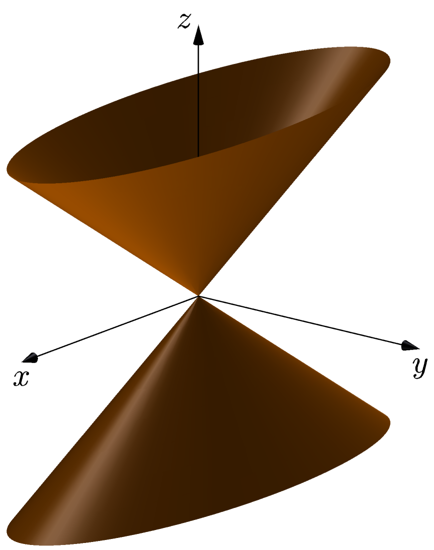
Obr. P.9: Dvoudílný hyperboloid

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=3,b=1,c=2;
triple f(pair z){
  real u=z.x, v=z.y;
  return (a*cos(u)*cos(v),b*cos(u)*sin(v),c*sin(u));
}

draw(surface(f,(-pi/2,-pi),(pi/2,pi),Spline,nu=8),
  orange);

axes3("$x$","$y$","$z$",(0,0,0),(a+1,b+2,c+1),Arrow3);

shipout(options="3Drender=SolidOutline");
```



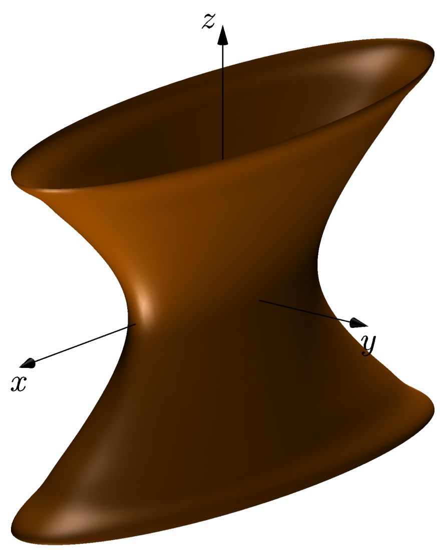
Obr. P.10: Elipsoid

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=3,b=1,c=2;
triple f(pair z){
  real u=z.x, v=z.y;
  return (a*(u/c)*cos(v),b*(u/c)*sin(v),u);
}

draw(surface(f,(-2,0),(2,2pi),Spline,nu=8),
     orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



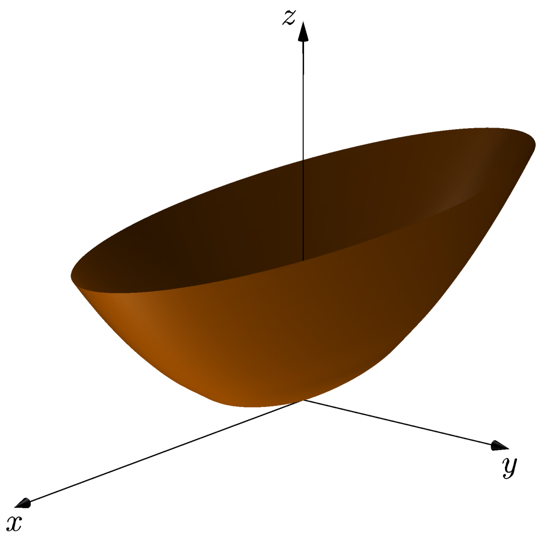
Obr. P.11: Eliptická kuželová plocha

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=3,b=1,c=2;
triple f(pair z){
  real u=z.x, v=z.y;
  return (a*sqrt(1+u^2)*cos(v),b*sqrt(1+u^2)*sin(v),c*u);
}

draw(surface(f,(-2,0),(2,2pi),Spline,nu=16),
     orange);

axes3("$x$","$y$","$z$",(0,0,0),(7,4,6),Arrow3);

shipout(options="3Drender=SolidOutline");
```



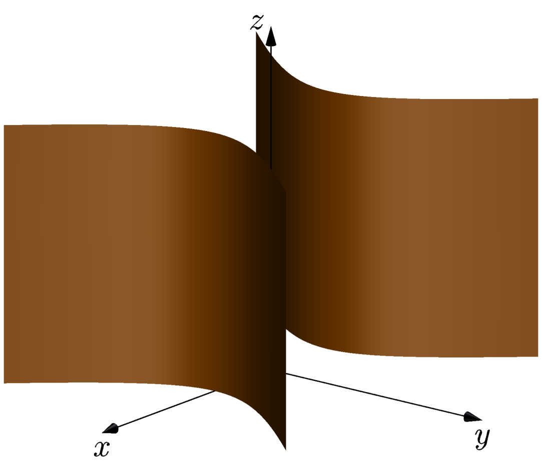
Obr. P.12: Eliptický hyperboloid

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=3,b=1;
triple f(pair z){
  real u=z.x, v=z.y;
  return (a*sqrt(u)*cos(v),b*sqrt(u)*sin(v),u);
}

draw(surface(f,(0,0),(3,2pi),Spline,nu=10),
     orange);

axes3("$x$","$y$","$z$",(0,0,0),(7,4,6),Arrow3);

shipout(options="3Drender=SolidOutline");
```



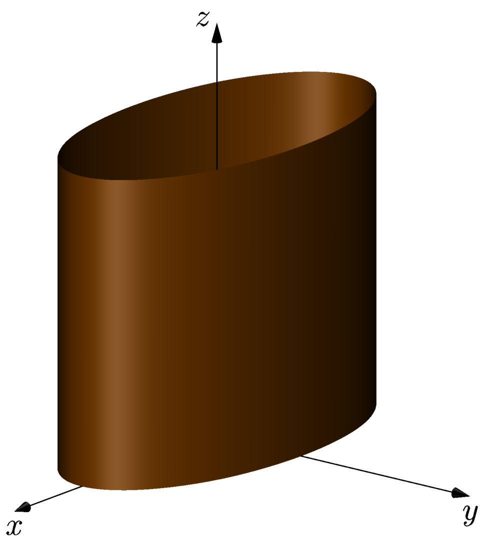
Obr. P.13: Eliptický paraboloid

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=1,b=1;
real f(real x){
  return a*sqrt(1+x^2/b^2);
}

path branch = graph(f,-2,2,n=10,join=operator ..);
surface s = extrude(branch,(0,0,3));

draw(rotate(-90,(0,0,1))*s,orange);
draw(rotate(90,(0,0,1))*s,orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,4),Arrow3);

shipout(options="3Drender=SolidOutline");
```



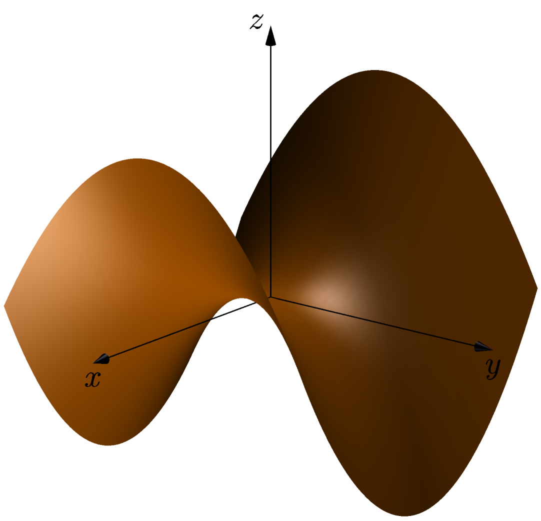
Obr. P.14: Hyperbolická válcová plocha

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real a=2,b=1;
real x(real t){ return a*cos(t);}
real y(real t){ return b*sin(t);}

path elipsa = graph(x,y,0,2pi,n=10,join=operator..)..cycle;

draw(extrude(elipsa,(0,0,3)),orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,4),Arrow3);

shipout(options="3Drender=SolidOutline");
```



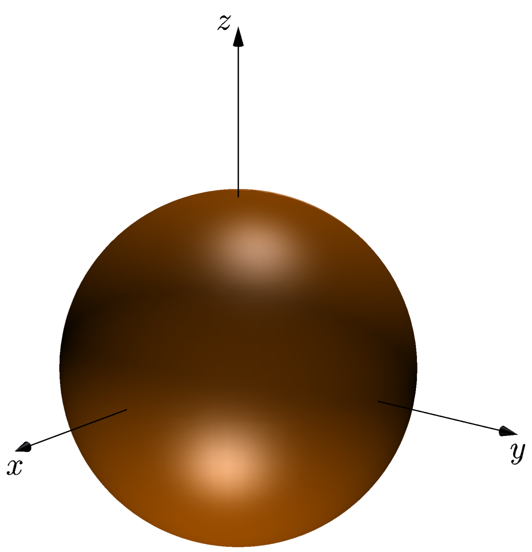
Obr. P.15: Válcová plocha

```
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real f(pair z){
  return z.x^2/2-z.y^2/2;
}

draw(surface(f,(-2,-2),(2,2),xsplinetype=Spline,nx=5),
     orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



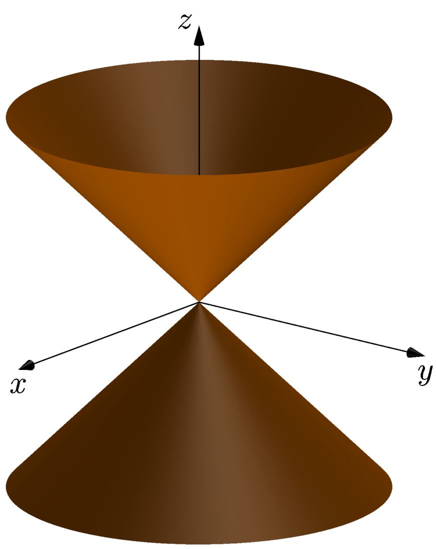
Obr. P.16: Hyperbolický paraboloid

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

draw(surface(sphere(1)),orange);

axes3("$x$","$y$","$z$",(0,0,0),(2,2,2),Arrow3);

shipout(options="3Drender=SolidOutline");
```



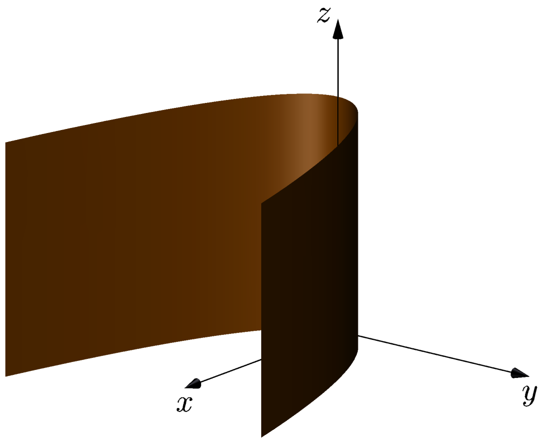
Obr. P.17: Koule

```
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real f(real x){
  return x;
}

draw(surface(revolution(f,-2,2,Z,n=10)),orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



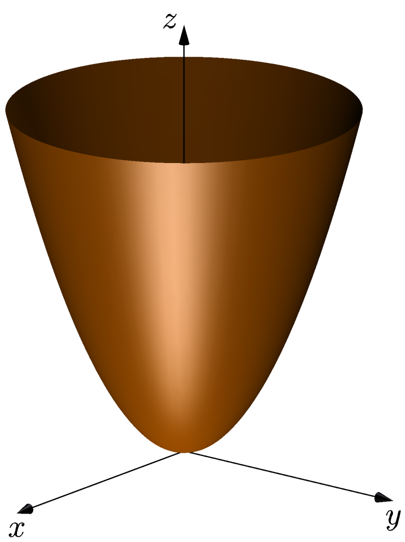
Obr. P.18: Kuželová plocha

```
import graph;
import graph3;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real p=1;
real f(real x){
  return x^2/p;
}

path parab = graph(f,-2,2,n=10,join=operator..);

draw(rotate(-90,(0,0,1))*extrude(parab,(0,0,3)),orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,4),Arrow3);

shipout(options="3Drender=SolidOutline");
```



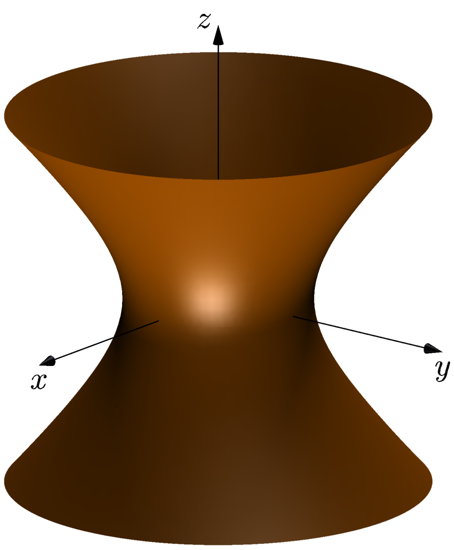
Obr. P.19: Parabolická válcová plocha

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real f(real x){
  return x^2;
}

draw(surface(revolution(f,0,2,n=8,operator..)),orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,5),Arrow3);

shipout(options="3Drender=SolidOutline");
```



Obr. P.20: Paraboloid

```
import graph3;
import solids;

size(200);
currentlight = White;
currentprojection = orthographic(5,4,2);

real f(real z){
  return sqrt(1+z^2);
}

draw(surface(rotate(90,(0,1,0))
     *revolution(f,-2,2,n=8,operator..,X)),orange);

axes3("$x$","$y$","$z$",(0,0,0),(3,3,3),Arrow3);

shipout(options="3Drender=SolidOutline");
```



Obr. P.21: Rotační hyperboloid

Literatura

[1] *Oficiální stránky Asymptote.* [online]. 2004 [cit. 2011-12-15]. Dostupné z WWW: http://asymptote.sourceforge.net/

[2] *Surface Parametrization of Nonsimply Connected Planar Bézier Regions.* [online]. 2011 [cit. 2011-12-15]. Dostupné z WWW: http://www.math.ualberta.ca/~bowman/publications/cad10.pdf

[3] *ASYMPTOTE - Galeries d'exemples.* [online]. 2011 [cit. 2011-12-15]. Dostupné z WWW: http://marris.org/asymptote/

[4] PLCH, Roman - ŠARMANOVÁ, Petra. *Interaktivní 3D grafika v HTML a PDF dokumentech.* Zpravodaj Československého sdružení uživatelů TEXu, Praha : Československé sdružení uživatelů TEXu, 18, 1-2, od s. 76-92, 16 s. ISSN 1211-6661.2008.

[5] PLCH, Roman - ŠARMANOVÁ, Petra - SOJKA, Petr. *Integrální počet funkcí více proměnných.* Elportál: portál Masarykovy univerzity [online], Brno : Masarykova univerzita, 2009, 1, 160 s. ISSN 1802-128X. 2009.

[6] LOMTATIDZE, Lenka - PLCH, Roman. *Sázíme v LATEXu diplomovou práci z matematiky.* 1.vyd. Brno : Masarykova univerzita, 2003. 122 s. ISBN 80-210-3228-6.

[7] ŽÁRA, Jiří - BENEŠ, Bedřich - SOCHOR, Jiří - FELKEL, Petr. *Moderní počítačová grafika.* 2.vyd. Praha : Computer Press, 2005. 609 s. ISBN 80-251-0454-0.

[8] *Specifikace formátu PRC.* [online]. 2008 [cit. 2012-02-16]. Dostupné z WWW: http://livedocs.adobe.com/acrobat_sdk/9/Acrobat9_HTMLHelp/API_References/PRCReference/PRC_Format_Specification/

Seznam obrázků

Rejstřík