Abstract types

An *abstract type* represents groups of other types, but can never be instantiated on its own. For example, every integer and floating point number is a real number. Therefore, there is an abstract type `Real`, which encapsulates many other types, including `Float64`, `Float32`, `Int64` and `Int32`.

We can test if type `T` is part of an abstract type `V` using the sytax `T <: V`:

In [99]:

```
Float64 <: Real
```

Out[99]:

```
true
```

In [2]:

```
Float32 <: Real
```

Out[2]:

```
true
```

In [3]:

```
Int64 <: Real
```

Out[3]:

```
true
```

In [4]:

```
Int32 <: Real
```

Out[4]:

```
true
```

As a counter example, complex numbers are not real, and therefore `Complex` types are not `<: Real`:

In [101]:

```
C=typeof(1+im)   # returns Complex{Int64} or Complex{Int32}, depending on th
e machine


C <: Real    # Complex numbers are not real!
```

Out[101]:

```
false
```

Super types

Every type has one and only one super type, which is *always* an abstract type. The function `super` applied to a type returns its super type:

In [7]:

```
super(Int32)  # returns Signed, which represents all signed integers.
```

Out[7]:

```
Signed
```

In [102]:

```
super(Int64)  # all returns Signed
```

Out[102]:

```
Signed
```

In [8]:

```
super(Float32)
```

Out[8]:

```
AbstractFloat
```

In [9]:

```
super(Float64)
```

Out[9]:

```
AbstractFloat
```

An Abstract type also has a super type:

In [10]:

```
super(Real)
```

Out[10]:

```
Number
```

In [11]:

```
super(Number)
```

Out[11]:

```
Any
```

In [13]:

```
super(Signed)
```

Out[13]:

```
Integer
```

In [14]:

```
super(Integer)
```

Out[14]:

```
Real
```

In [15]:

```
super(UInt32)
```

Out[15]:

```
Unsigned
```

In [16]:

```
super(AbstractFloat)
```

Out[16]:

```
Real
```

In [17]:

```
super(Number)
```

Out[17]:

```
Any
```

Any is the largest type that contains every other type, and its super type is itself:

In [18]:

```
super(Any)
```

Out[18]:

```
Any
```

Abstract types and defining functions

We can use abstract types to define functions, restricting the definition to only apply for types which are subtypes of the abstract type. For example, the following defines differently depending on whether `x` is an `Integer` or an `AbstractFloat`.

In [1]:

```
function myfactorial(x::Integer)
    factorial(x)
end
function myfactorial(x::AbstractFloat)
    gamma(x+1)
end
```

Out[1]:

```
myfactorial (generic function with 2 methods)
```

In [2]:

```
myfactorial(5)   # 5 is an Int64, which is <: Integer, hence the first defin
ition is called
```

Out[2]:

```
120
```

In [3]:

```
myfactorial(UInt32(5))   # UInt32(5) is a UInt64, which is <: Integer, hence
 the first definition is called
```

Out[3]:

```
120
```

In [4]:

```
myfactorial(5.5)  # 5.5 is a Float64, which is <: AbstractFloat, hence the s
econd definition is called
```

Out[4]:

```
287.88527781504433
```

In [5]:

```
myfactorial(5.5f0)  # 5.5f0 is a Float32, which is <: AbstractFloat, hence t
he second definition is called
```

Out[5]:

```
287.88528f0
```

Abstract types and vectors

There are many different types that represent vectors. This property is captured using `AbstractVector`. For example, the command `rand(5)` returns a `Vector{Float64}`.

In [6]:

```
v=rand(5)
typeof(v)==Vector{Float64}
```

Out[6]:

```
true
```

On the other hand, the syntax `a:b` returns something that acts like a vector, but is not a vector:

In [7]:

```
r=2:6
r[2]
```

Out[7]:

```
3
```

In [8]:

```
typeof(r)==Vector{Int64}
```

Out[8]:

```
false
```

Both `v` and `r` are `AbstractVectors`:

In [9]:

```
typeof(v) <: AbstractVector{Float64}
```

Out[9]:

```
true
```

In [10]:

```
typeof(r) <: AbstractVector{Int64}
```

Out[10]:

```
true
```

Any `AbstractVector` can be converted to a `Vector` using the function `collect`:

In [11]:

```
r=2:7
collect(r)
```

Out[11]:

```
6-element Array{Int64,1}:
 2
 3
 4
 5
 6
 7
```

In [12]:

```
typeof(collect(r)) == Vector{Int64}
```

Out[12]:

```
true
```

Big difference: the entries of `Vectors` can be changed, but `AbstractVectors` cannot, in general.

In [13]:

```
v=rand(5)
v[2]=3
v
```

Out[13]:

```
5-element Array{Float64,1}:
 0.586114
 3.0
 0.0513032
 0.332904
 0.440058
```

In [14]:

```
r=2:6
r[2]=3   # Not allowed
```

```
LoadError: indexed assignment not defined for UnitRange{Int64}
while loading In[14], in expression starting on line 2

 in setindex! at abstractarray.jl:592
```

Use `collect` to make an `AbstractVector` changeable:

In [15]:

```
r=collect(2:6)
r[2]=3   # Allowed
r
```

Out[15]:

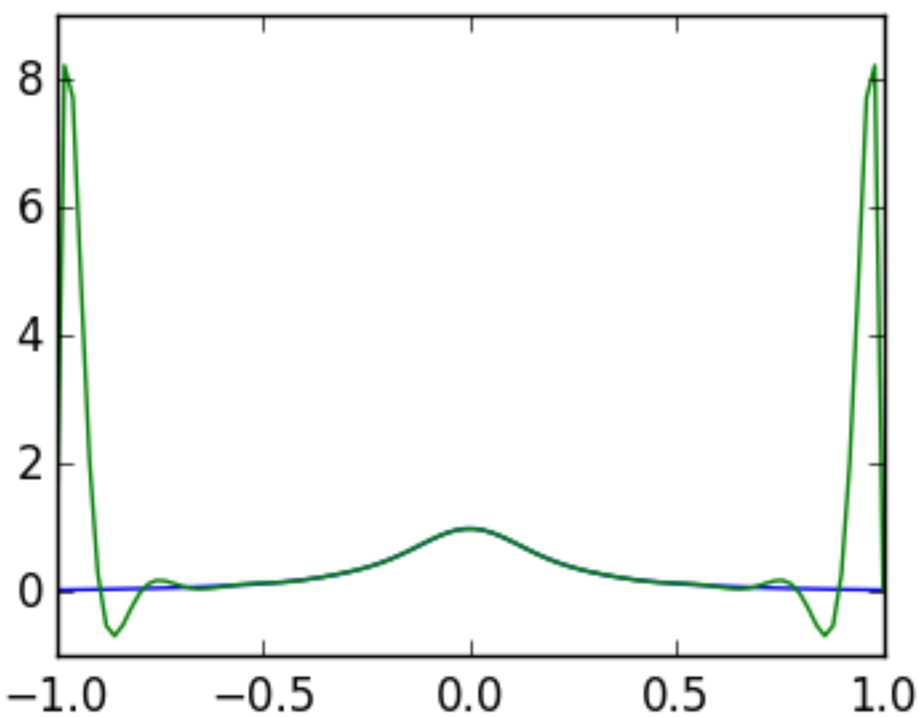
```
5-element Array{Int64,1}:
 2
 3
 4
 5
 6
```

Interpolation

We saw last lecture that interpolating at evenly spaced points, runs into issues:

In [16]:

```
using PyPlot


## this function evaluates a Taylor polynomial at a point x, or vector of po
ints x, where c is a vector of coefficients.
function p(c,x)
    n=length(c)
    ret=0.0
    for k=1:n
        ret=ret+c[k]*x.^(k-1)   # use .^ instead of ^ to allow x to be a vec
tor
    end
    ret
end


g=linspace(-1.,1.,100)    # plotting grid: 100 points between -1 and 1

plot(g,1./(25g.^2+1))     # plot the true function in blue

n=20                      # interpolate at 20 points
x=linspace(-1.,1.,n)
V=Float64[x[k]^(j-1) for k=1:n,j=1:n]    # construct the Vandermonde matrix

f=1./(25x.^2+1)          # evaluate the true function at the interpolation po
ints
c=V\f                    # calculate the coefficients of the interpolating po
lynomial
plot(g,p(c,g))           # plot the interpolating polynomial be evaluating at
 the plotting grid ;
```



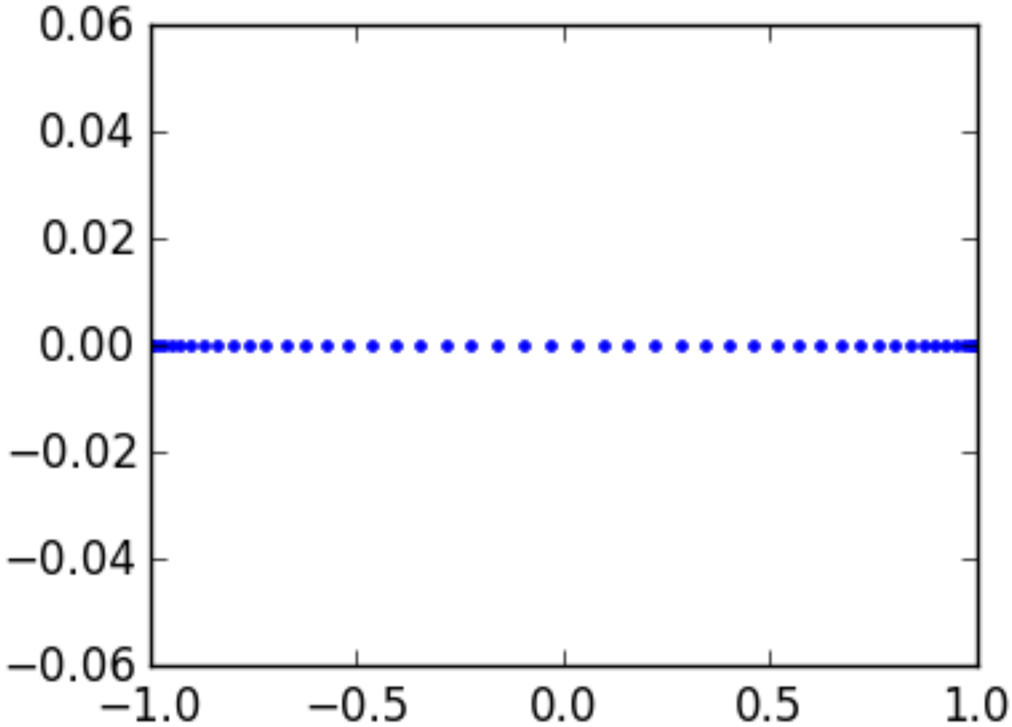
Out[16]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x30fda3410>
```

The "best" way would be to choose different interpolation points. A particularly good choice is as follows:

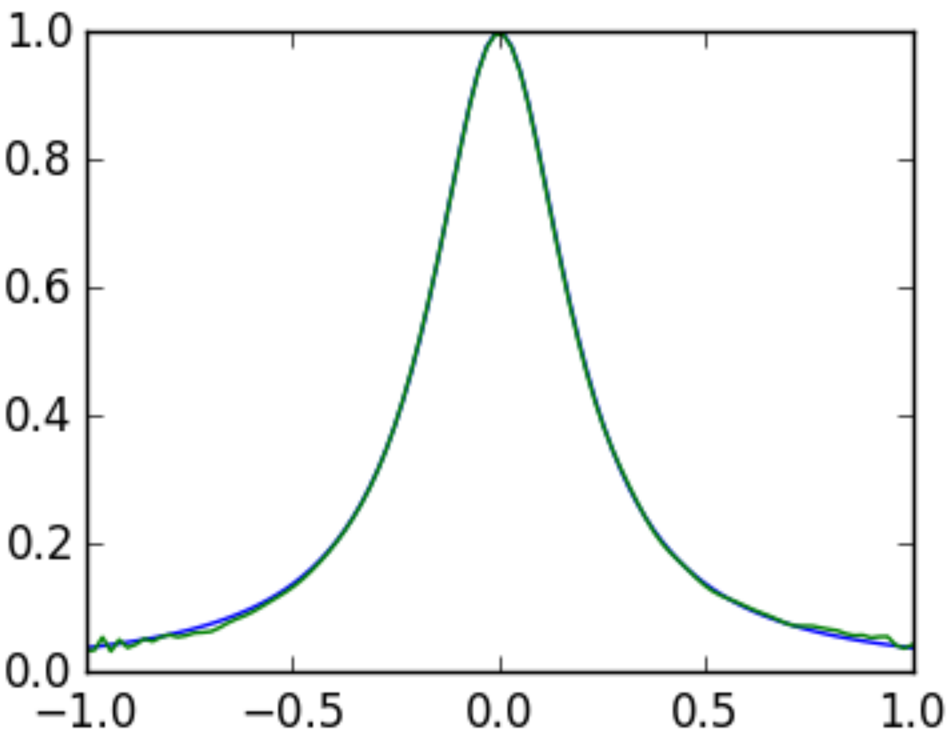
In [25]:

```
n=50
θ=linspace(0,π,n)
x=cos(θ)

plot(x,zeros(n);marker=".",linestyle="");
```



In [24]:

```
n=100
θ=linspace(0,π,n)
x=cos(θ)   # this is my new x, called "Chebyshev points"


V=Float64[x[k]^(j-1) for k=1:n,j=1:n]
f=1./(25x.^2+1)
c=V\f   # calculate the new coefficients at new points


plot(g,1./(25g.^2+1))
plot(g,p(c,g));
```



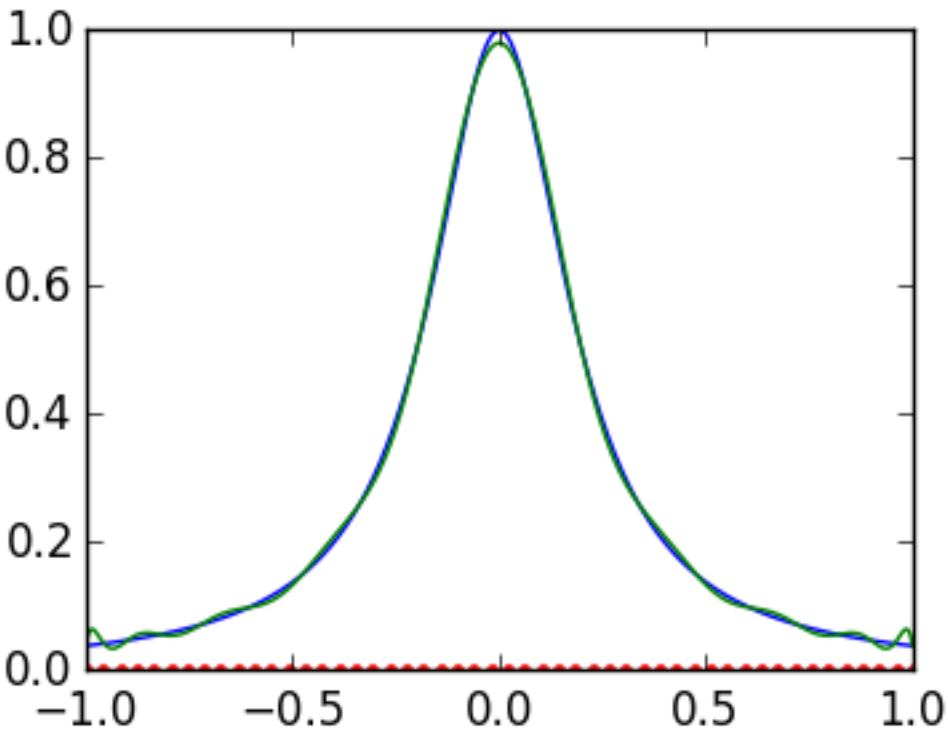
Note there are still issues: we fixed the grid, but we also have to choose a better basis. We will come back to this later.

Least squares

But what if we only know the function at evenly spaced points? We can still reliably approximate the function by using more points than coefficients.

In [29]:

```
g=linspace(-1,1,1000)

plot(g,1./(25g.^2+1))  # plot the true function

m=20   # number of coefficients
n=50  # number of points

x=linspace(-1.,1.,n)   # use n evenly spaced points
V=Float64[x[k]^(j-1) for k=1:n,j=1:m]    # make an n x m Vandermonde matrix

f=1./(25x.^2+1)     # f evaluated at the m points
c=V\f    # find c so that V*c is approximately f


plot(g,p(c,g))
plot(x,zeros(n);marker=".",linestyle="")  # plot the grid;
```



\ only solves approximately: V*c is not exactly f:

In [23]:

```
norm(V*c-f)
```

Out[23]:

```
0.05488784046826881
```

Increasing the coefficients and number of points appropraitely improves the accuracy:

In [26]:

```
g=linspace(-1,1,1000)

plot(g,1./(25g.^2+1))  # plot the true function

m=80   # number of coefficients
n=m^2  # number of points


x=linspace(-1.,1.,n)   # use n evenly spaced points
V=Float64[x[k]^(j-1) for k=1:n,j=1:m]    # make an n x m Vandermonde matrix

f=1./(25x.^2+1)     # f evaluated at the m points
c=V\f    # find c so that V*c is approximately f

plot(g,p(c,g))
plot(x,zeros(n);marker=".",linestyle="");
```