README

March 6, 2022

1 Configurations

Nightly toolchain is needed for `slas` to work, which is a linear algebra system written by me.

Plotly and itertools is also needed for plotting data.

[2]:
```
:toolchain nightly

:dep slas = { git = "https://github.com/unic0rn9k/slas", features =
 ↪["fast-floats"] }
:dep plotly
:dep itertools-num

#![allow(incomplete_features)]
#![feature(generic_const_exprs, test, const_fn_floating_point_arithmetic)]

use slas::prelude::*;
use slas_backend::*;

extern crate plotly;
extern crate rand_distr;
extern crate itertools_num;
extern crate itertools;

use itertools_num::linspace;
use plotly::common::{
    ColorScale, ColorScalePalette, DashType, Fill, Font, Line, LineShape,
 ↪Marker, Mode, Title,
};
use plotly::layout::{Axis, BarMode, Layout, Legend, TicksDirection};
use plotly::{Bar, NamedColor, Plot, Rgb, Rgba, Scatter};
use rand_distr::{Distribution, Normal, Uniform};
```

[2]:
```
Toolchain: nightly
```

2 Some example code

For some reasone slas vectors needs type annotations when used in evcxr (Jupyter), even though this is not required normally…

[3]:
```
let mut a: StaticCowVec<f32, 3> = moo![f32: 1, 2, 3];
let b: StaticCowVec<f32, 3> = moo![f32: 0..3];

println!("{}", b.static_backend::<Blas>().dot(&a.static_backend::<Blas>()));
```

```
8
```

[4]:
```
println!("{a:?}");
```

```
[1.0, 2.0, 3.0]
```

3 Function for plotting

first we define a function for plotting a vector with index on the x-axis and values on the y-axis

[5]:
```
extern crate serde;

fn plot_vector<const LEN: usize>(v: StaticVecRef<f32, LEN>){
    let t: Vec<f64> = linspace(0., LEN as f64, LEN).collect();
    let trace = Scatter::new(t, **v).mode(Mode::Markers);
    let mut plot = Plot::new();
    plot.add_trace(trace);
    let layout = Layout::new().height(v.iter().map(|n|*n as usize).max().
  ↪unwrap());
    plot.set_layout(layout);
    plot.notebook_display();
}
```

3.1 Generating and plotting an example wave

[6]:
```
use std::f32::consts::PI;

const LEN: usize = 2_usize.pow(10);

// Fast floats are more inacurrate, but faster to do operations on then regular␣
  ↪f32 and f64 floats.
// When you see something like fast(PI), it means a fast float is being used␣
  ↪instead of a regular one.
use fast_floats::*;

let example: StaticCowVec<f32, LEN> = moo![|t|{
    let t = fast(t as f32) * fast(PI) / fast(512.);
```

```
    *((t * 22.).sin_() + (t * 25.).sin_())
}; LEN];

plot_vector(example.moo_ref());
```

4 Euler's identity

This is just an example that shows complex math using slas works.

[7]:
```
im(std::f32::consts::PI).exp_()
```

[7]: (-1 - 0.00000008742278im)

5 Cooley-Tukey in pseudo-code

```
FFT(x){
    if x.len < 2 then return x

    even = FFT(x[%2==0])
    odd  = FFT(x[%2==1])

    k = range(0, len / 2)

      = [e^(-2im * np.pi * k / x.len) * odd[k]]

    return [even[k] +  [k]].append([even[k] -  [k]])
}
```

6 Cooley-Tukey in Rust

This implementation of the cooley-tukey FFT algorithm is entirely statically allocated. Theoretically it should be able to just do one allocation (besides pointers), as it uses a return vector, allocated when calling the outer function, to store all temporary values, besides pointers.

[14]:
```
// x    = pointer to input vector.
//
// o    = pointer to output vector.
//
// n  = length of ipnut and output vectors.
//      (This function will not check if the length is valid,
//       Which is why `fft` should always be used instead.)
//
// ofset = 2 ^ recursion depth (starts at 0).
//
// LEN = length of original input vektor.
//
```

```rust
// n = LEN / ofset
unsafe fn unsafe_fft<const LEN: usize>(x: *const Complex<f32>, o: *mut Complex<f32>, n: usize, ofset: usize)
    -> *const Complex<f32>
{
    if n < 2{
        return x
    }

    // Recursively compute even and odd transformed coefficients.
    // The even transformed coefficients will be stored at the right hand side of the vector,
    // hence why o.add(len/2) is used as the output pointer for the function call.
    let even = unsafe_fft::<LEN>(x, o.add(n/2), n/2, ofset * 2);

    // ofset is added to the input pointer, so the memory read from is shiftet to the right,
    // This is done because 0 is even.
    let odd = unsafe_fft::<LEN>(x.add(ofset), o, n/2, ofset * 2);

    for j in 0..n/2{
        // Memory address where the k'th even transformed coefficients is stored
        // (This is calculated by shifting the `even` pointer k*size_of(f32) to the right)
        let even = *even.add(j);

        // Memory address where the k'th odd transformed coefficients is stored
        let odd = *odd.add(j);

        // This will compute the n'th root of unity ( ),
        // which will temporarily be stored in the location of the positive twiddle factor.
        *o.add(j) = im(-2. * PI * j as f32 / n as f32).exp_();

        // After the negative transformed coefficients is calculated, as to not overwrite  ,
        // which is also needed to compute the positive transformed coefficients.
        *o.add(j+n/2) = even - *o.add(j) * odd;

        // Positive transformed coefficients is computed.
        *o.add(j) = even + *o.add(j) * odd;
    }

    o
```

```
}

fn fft<const LEN: usize>(x: StaticVecRef<Complex<f32>, LEN>) -> [Complex<f32>;␣
↪LEN]{
    assert_eq!(LEN & (LEN - 1), 0);
    let mut ret = **x;
    unsafe{ unsafe_fft::<LEN>(x.as_ptr(), ret.as_mut_ptr(), LEN, 1) };
    ret
}
```

[16]:
```
let a = [re(0.), re(1.), re(2.), re(3.)];
let b = fft(a.moo_ref());

println!("{:#?}", b);

// Expected result:
//   6 + 0im
//  -2 + 2im
//  -2 + 0im
//  -2 - 2im
```

```
[
    (6 + 0im),
    (-1.9999999 + 2im),
    (-2 + 0im),
    (-2 - 2im),
]
```

[18]:
```
// Create some empty vectors to hold our wave sample,
// and fourier transformed result.
let mut y: [f32; LEN] = [0f32; LEN];
let mut y_hat: [Complex<f32>; LEN] = [re(0f32); LEN];

// Add two sinus waves together and write them to y and y_hat.
for i in 0..y_hat.len(){
    // t (time) is equal to i (index) times pi divided by the sample rate.
    // The sample rate could be pretty much any number, results may vary though.
↪..
    let t = fast(i as f32) * fast(PI) * 2. / fast(LEN as f32);

    // y(t) = sin(t * f_1) + sin(t * f_2) ... + sin(t * f_n)
    // In this example Hertz is just a made up unit,
    // there really is no units on any of these numbers.
    y[i] = *((t * 14.).sin_() + (t * 19.).sin_() + (t * 7.).sin_());
    y_hat[i] = re(y[i]);
}
```

```
plot_vector(y.moo_ref());
```

[19]:
```
// Fourier Magic...
y_hat = fft(y_hat.moo_ref());

// Copy the real part of all the fourier transformed points to a new vector.
for i in 0..y.len(){
    y[i] = y_hat[i].re.abs()
}
```

[19]:
```
()
```

[20]:
```
plot_vector(y[0..80].moo_ref::<80>());
// Here only the points from 0 to 80 are plotted.
// This is just to make the relevant peaks more visible.
//
// The frquencies used above for generating the y wave,
// should be the same as the labeled x component shown on the graph bellow.
// You can hover you mouse over the points on the graph to show theri
 ↪cordinates :)
```