Lorenz Gauge and retarded potentials

written by 110 on Functor Network
original link: https://functor.network/user/2029/entry/1135

Everybody knows about Maxwell's equations, and almost more people know about gauge. In electromagnetism we can define two potentials usually noted ϕ and $\mathbf{A}$ from which we can derive the electric and magnetic field.

$$\mathbf{B} = \nabla \times \mathbf{A}$$

$$\mathbf{E} = -\nabla\phi - \frac{\partial \mathbf{A}}{\partial t}$$

The funny thing about those potentials is that they are not uniquely defined. The way potentials are computed varies depending on what is called a gauge fixing.

Two notables gauge are the Coulomb gauge and the Lorenz gauge.

In the Lorenz gauge, potentials can be expressed using localised retarded charges and currents

$$\varphi(\mathbf{r}, t) = \frac{1}{4\pi\epsilon_0} \int \frac{\rho(\mathbf{r}', t')}{|\mathbf{r} - \mathbf{r}'|} \, \mathrm{d}^3\mathbf{r}'$$

$$\mathbf{A}(\mathbf{r}, t) = \frac{\mu_0}{4\pi} \int \frac{\mathbf{J}(\mathbf{r}', t')}{|\mathbf{r} - \mathbf{r}'|} \, \mathrm{d}^3\mathbf{r}'$$

What "retarded potentials" means here is not a pun for witty people to mock the slower half, (while slowing themselves with that sort of puns), it means the fields in point $\mathbf{r}$ at time t are the results of the charges and currents at point $\mathbf{r}'$ and time $t' = t - \frac{\|\mathbf{r}-\mathbf{r}'\|}{c}$, hence the name "retarded". Which is a very comfortable interpretations of fields behaviors. If I remember correctly it also allows for advanced potentials, but that's another topic I might never be able to dig in unfortunately.

In the Coulomb gauge on the opposite, the potentials are expressed using instantaneous charges and fields in the whole space.

$$\varphi(\mathbf{r}, t) = \frac{1}{4\pi\varepsilon_0} \int \frac{\rho(\mathbf{r}'', t)}{R} d^3\mathbf{r}''$$

$$\mathbf{A}(\mathbf{r}, t) = \nabla \times \int \frac{\mathbf{B}(\mathbf{r}'', t)}{4\pi R} d^3\mathbf{r}''$$

Here $\mathbf{r}''$ is an instantaneaous position (in the inertial frame of the observer). The Coulomb gauge gives a taste of non locality. It looks very odd that the fields at one specific point in space and time could be defined by the charges and fields of the whole universe at that very same instant. It is actually deterministic as

hell and one can reconcile the local vision of the Lorenz gauge with the non local vision of the Coulomb gauge through a sort of "transitivity" idea, each instantaneous field and charge being the result of sources located elsewhere in the past. Still, it feels a bit weird, notably because it does not include other interactions, what's in electromagnetism stays in electromagnetism.

Another fascinating stuff about retarded potentials is that it implies all fields can be seen as the vector sum of many fields created by (almost) uncontable sources, while we are used to the local version of electromagnetism where all fields are created by nearby fields. This is particularly striking for experiments like diffraction. The classical description of diffraction involves a plane wave hitting a drilled wall that creates a new wave originating at the hole location. If the (almost) plane wave was created by a very remote point source, the local description of fields make it look like a wave turning around an obstacle. In the retarded picture, the local fields are only the superposition of many fields each of which having an origin at the other end of a straight line. The fields do not really follow curved path or broken lines ine that picture.

Those observations lead me to believe that a program relying on retarded potentials could be much faster than finite differences or finite element methods because there would be no need to really compute the field in the vacuum, and you could only care about the sources and the absorbers.

Unfortunately, I am too much of a bad programmer, and I did not manage to find a way to efficiently implement that idea. I stay convinced it's doable.

```
mod methods;
use crate::methods::*;

const WIDTH: usize = 640;
const HEIGHT: usize = 480;

fn main()
{
    //let buffer_len=1_000;
    //let mut buffer: Vec<Vec<u32>> = vec![vec![0; WIDTH * HEIGHT];buffer_len];
        //prototype buffer
        /*
        for j in 0..buffer_len
        {
            for i in 0..WIDTH*HEIGHT
            {
                //buffer[i] = (( i as u32)%255)*( 1<<(8*((i/256)%3)) );
                buffer[j][i] = (i as u32 + 100*(j as u32) )&0x00_00_FF_FF;
                //buffer[i] = i as u32;
            }
        }
        // */
    let duration = 400;
```

```
    let electric_field = field_value(WIDTH, HEIGHT, duration);
    let buffer = field_to_buffer(&electric_field);

    display(&buffer,WIDTH,HEIGHT);
}

use std::{thread, time};
use minifb::{Key, Window, WindowOptions};
use rayon::prelude::*;
//use num_traits::{Zero,One};

pub fn display(buffer: &Vec<Vec<u32>>, width: usize, height: usize){
    let mut window = Window::new(
        "Test - ESC to exit",
        width,
        height,
        WindowOptions::default(),
    ).unwrap_or_else(|e| { panic!("{}", e);});

    // Limit to max ~60 fps update rate
    window.limit_update_rate(Some(time::Duration::from_micros(16_600)));

    while window.is_open() && !window.is_key_down(Key::Escape)
    {
        for j in 0..buffer.len()
        {
            window.update_with_buffer(&(buffer[j]), width, height).unwrap();
            thread::sleep(time::Duration::from_micros(16_600));
        }
    }

}
pub fn radius(width: usize, height: usize)->(Vec<Vec<f64>>,Vec<Vec<f64>>,Vec<Vec<f64>>,Vec<V
{
    let mut radius_array= vec![vec![0.0;width]; height];
    let mut inverse_radius_array= vec![vec![0.0;width]; height];
    let mut inverse_radius_square_array= vec![vec![0.0;width]; height];
    let mut n_x= vec![vec![0.0;width]; height];
    let mut n_y= vec![vec![0.0;width]; height];

    let mut x_2;
    let mut y_2;
    let mut r_2;
```

```
    for i in 0..width
    {
        x_2 = (i*i) as f64;

        for j in 0..height
        {
            y_2 = (j*j) as f64;
            r_2 = x_2+y_2;

            radius_array[j][i]=r_2.sqrt();
            if r_2 !=0.0
            {
                inverse_radius_array[j][i]=radius_array[j][i].powf(-1.0);
                inverse_radius_square_array[j][i]=inverse_radius_array[j][i].powf(2.0);

                n_x[j][i]= (i as f64)*inverse_radius_array[j][i];
                n_y[j][i]= (j as f64)*inverse_radius_array[j][i];
            }else
            {
                inverse_radius_array[j][i]=0.0;
                inverse_radius_square_array[j][i]=0.0;
                n_x[j][i]=0.0;
                n_y[j][i]=0.0;
            }
        }
    }
    return (radius_array,inverse_radius_array,inverse_radius_square_array,n_x,n_y);
}


pub fn motion(width: usize, height: usize, duration:usize)->(Vec<i32>,Vec<i32>,Vec<f64>,Vec<
{
    let mut x: Vec<i32> = vec![0;duration];
    let mut y: Vec<i32> = vec![0;duration];
    let mut v_x: Vec<f64> = vec![0.0;duration];
    let mut v_y: Vec<f64> = vec![0.0;duration];
    let mut a_x: Vec<f64> = vec![0.0;duration];
    let mut a_y: Vec<f64> = vec![0.0;duration];
    let mut r_cos;
    let mut r_sin;
    let omega= 1.0/40.0;
    let circle_radius= (height as f64)/8.0;
    for t in 0..duration
    {
        r_cos= (circle_radius*((t as f64)*omega).cos()).round() as i32;
```

```
        r_sin= (circle_radius*((t as f64)*omega).sin()).round() as i32;
        x[t]=(width  as i32)/2+ r_cos;
        y[t]=(height as i32)/2+ r_sin;
        v_x[t]=-(r_sin as f64)*omega;
        v_y[t]=(r_cos as f64)*omega;
        a_x[t]=-v_y[t]*omega;
        a_y[t]=v_x[t]*omega;
        //println!("v_x:{}",v_x[t]);
        //println!("v_y:{}",v_y[t]);
        //println!("a_x:{}",v_x[t]);
        //println!("a_y:{}",v_y[t]);


    }
    return(x,y,v_x,v_y,a_x,a_y);
}

pub fn average<T>( array : & mut Vec<Vec<T>>)
where
    T: num::Zero+ num::One+ std::ops::Div<Output = T> + std::ops::AddAssign+Clone
    //T: num::* + std::ops::Div<Output = T> + std::ops::AddAssign+Clone

{
    let width = (*array)[0].len();
    let height = (*array).len();
    let mut  c:T =T::zero();
    let mut av:T =T::zero();
    for i in 1..width-1
    {
        for j in 1..height-1
        {
            if (*array)[j][i].is_zero()
            {
                for k in 0..9
                {
                    if !(*array)[j-1+k/3][i-1+k%3].is_zero()
                    {
                        c+= T::one();
                        av= av +((*array)[j-1+k/3][i-1+k%3]).clone();
                    }

                }
                av = av/c;
                (*array)[j][i]=av;
                av= T::zero();
                c = T::zero();
```

```
            }
        }
    }

}
/*
pub fn average( array : & mut Vec<Vec<f64>>)
{
    let width = (*array)[0].len();
    let height = (*array).len();
    let mut c =0.0;
    let mut av=0.0;
    for i in 1..width-1
    {
        for j in 1..height-1
        {
            if (*array)[j][i]==0.0
            {
                for k in 0..8
                {
                    if (*array)[j-1+k/3][i-1+k%3]!=0.0
                    {
                        c+=1.0;
                        av+=(*array)[j-1+k/3][i-1+k%3];
                    }

                }
                av = av/c;
                (*array)[j][i]=av;
                av= 0.0;
                c = 0.0;
            }
        }
    }

}
// */
pub fn field_value(width: usize, height: usize, duration: usize) -> Vec<Vec<Vec<f64>>> {
    let mut electric_field = vec![vec![vec![0.0; width]; height]; duration];
    let motion = motion(width, height, duration);
    let radius = radius(width, height);

    // Pre-compute beta values for all timesteps
    let c = 5;
    let one_over_c = 1.0 / (c as f64);
    let betas: Vec<(f64, f64, f64, f64)> = (0..duration).into_par_iter().map(|t| {
```

```
    let beta_x = motion.2[t] * one_over_c;
    let beta_y = motion.3[t] * one_over_c;
    let beta_dot_x = motion.4[t] * one_over_c;
    let beta_dot_y = motion.5[t] * one_over_c;
    (beta_x, beta_y, beta_dot_x, beta_dot_y)
}).collect();

// Pre-compute position values
let positions: Vec<(i32, i32)> = motion.0.iter().zip(motion.1.iter())
    .map(|(&x, &y)| (x, y))
    .collect();

// Process each timestep in parallel
(0..duration).into_par_iter().for_each(|t| {
    let (pos_x, pos_y) = positions[t];
    let (beta_x, beta_y, beta_dot_x, beta_dot_y) = betas[t];
    let beta_x_2 = beta_x * beta_x;
    let beta_square = beta_x_2 + beta_y * beta_y;
    let gamma_2 = 1.0 - beta_square;

    for i in 1..width-1 {
        let x = i as i32 - pos_x;
        let x_2 = x * x;

        for j in 1..height-1 {
            let y = j as i32 - pos_y;
            let y_2 = y * y;
            let r_2 = x_2 + y_2;

            if r_2 == 0 { continue; }  // Skip calculation at source point

            let r = (r_2 as f64).sqrt().round() as usize;
            let future_t = t + r/c;
            if future_t >= duration { continue; }  // Skip if beyond duration

            //*
            // Use pre-computed n_x and n_y from radius arrays
            let n_x = if x < 0 {
                -radius.3[y.abs() as usize][x.abs() as usize]
            } else {
                radius.3[y.abs() as usize][x.abs() as usize]
            };

            let n_y = if y < 0 {
                -radius.4[y.abs() as usize][x.abs() as usize]
            } else {
```

```
                    radius.4[y.abs() as usize][x.abs() as usize]
                };
                // */
                let scalar_product = n_x * beta_x + n_y * beta_y;
                if (1.0 - scalar_product).abs() < 1e-10 { continue; }  // Skip near-singular
                let static_scale = 0.005;
                let factor = (1.0 - scalar_product).powf(-3.0);

                // Complete expression with both non-radiative and radiative parts
                let field_value = factor * (
                    static_scale*gamma_2 * (n_y - beta_y) * radius.2[y.abs() as usize][x.abs
                    one_over_c * radius.1[y.abs() as usize][x.abs() as usize] *
                    (n_x * ((n_y - beta_y) * beta_dot_x - (n_x - beta_x) * beta_dot_y))
                );

                // Use atomic operation to safely update the future time
                use std::sync::atomic::{AtomicU64, Ordering};
                let field_bits = field_value.to_bits();
                let atomic_ptr = &electric_field[future_t][j][i] as *const f64 as *const Ato
                unsafe {
                    (*atomic_ptr).store(field_bits, Ordering::Relaxed);
                }
            }
        }
    });

    // Apply averaging after all calculations
    for t in 0..duration {
        average(&mut electric_field[t]);
    }

    electric_field
}

/*
pub fn field_value(width: usize, height: usize, duration:usize)->Vec<Vec<Vec<f64>>>
{
    let mut electric_field:Vec<Vec<Vec<f64>>> = vec![vec![vec![0.0;width];height];duration];
    let motion = motion(width,height,duration);
    let radius = radius(width,height);
    let mut x;
    let mut y;
    let mut pos_x;
    let mut pos_y;
    let mut r;
    let mut n_x;
```

```
    let mut n_y;
    let c= 10;
    let one_over_c= 1.0/(c as f64);

    for t in 0..duration
    {
        for i in 0..width
        {
            pos_x=motion.0[t];
            //x=i as i32 - (pos_x-(width/2) as i32);
            x=i as i32 - pos_x;
            for j in 0..height
            {
                pos_y=motion.1[t];
                //y= j as i32 - (pos_y-(height/2) as i32);
                y= j as i32 - pos_y;

                r= (( ( x*x + y*y ) as f64).sqrt()).round() as usize;
                if t + r/c  <duration
                {
                    electric_field[t+r/c][j][i]=radius.1[ (y.abs()) as usize ][ (x.abs()) as

                }
            }
        }
        average(& mut (electric_field[t]));
    }
    electric_field
}
// */

pub fn field_to_buffer(electric_field: &Vec<Vec<Vec<f64>>>)->Vec<Vec<u32>>
{
    let duration = (*electric_field).len();
    let height = (*electric_field)[0].len();
    let width = (*electric_field)[0][0].len();

    let mut buffer:Vec<Vec<u32>>=vec![vec![0; width*height];duration];
    let mut max_value=0.0;
    let mut min_value=0.0;

    //search the max value and the min value;
    for t in 0..duration
    {
        for i in 0..width
        {
```

```
            for j in 0..height
            {
                if electric_field[t][j][i]>max_value
                {
                    //println!("max value:{}", max_value);
                    max_value= electric_field[t][j][i];
                }
                if electric_field[t][j][i]<min_value
                {
                    min_value= electric_field[t][j][i];
                }
            }
        }
    }

    //normalisation
    if max_value < -min_value
    {
        max_value = -min_value;
    }
    println!("max value:{}", max_value);


    for t in 0..duration
    {
        for j in 0..height
        {
            for i in 0..width
            {
                if electric_field[t][j][i]> 0.0
                {
                    buffer[t][i+j*width ] = ( ( ( (electric_field[t][j][i]/max_value)*65536.
                    //buffer[t][i+j*width ] = ( ( (electric_field[t][j][i])*65536.0*64.0/0.0
                }else if electric_field[t][j][i]< 0.0
                {
                    buffer[t][i+j*width ] = ( ( ( (-electric_field[t][j][i]/max_value)*256.0
                    //buffer[t][i+j*width ] = (( (-electric_field[t][j][i])*256.0*64.0/0.000
                }

            }
        }
    }
    buffer

}
```

put the main.rs file in a src directory. create a methods directory at the same level than the src directory and add the file mod.rs to it add the Cargo.toml file at the same level than those two directories.

```
[package]
name = "lienard_potential"
version = "0.1.0"
edition = "2021"

[dependencies]
minifb = "0.25.0"
num = "0.4.1"
num-traits = "0.2.18"
rayon = "1.10.0"

[profile.release]
codegen-units = 1
lto = "fat"
```

cargo run –release