Integer multiplication in time $O(n \log n)$
DD2467 Individual Project in TCS

Marcus Östling

June 1, 2022

1 Paper

`https://hal.archives-ouvertes.fr/hal-02070778/document`

My implmenetation so far: `https://github.com/mackeper/integer-multiplication`, not the cleanest code at the time.

2 Gaussian resampling

Goal:
$\mathcal{A} : \otimes_i C^{s_i} \to \otimes_i C^{t_i}$
$\mathcal{B} : \otimes_i C^{t_i} \to \otimes_i C^{s_i}$

Such that:
$\mathcal{F}_{s_1,\ldots,s_d} = 2^{\gamma} \mathcal{B} \mathcal{F}_{t_1,\ldots,t_d} \mathcal{A}$

Approximation:
$\tilde{\mathcal{A}} := \tilde{\mathcal{S}}'$

$\tilde{\mathcal{B}} := \tilde{\mathcal{P}}_s^{-1} \tilde{\mathcal{D}}' \tilde{\mathcal{J}}' \tilde{\mathcal{C}} \tilde{\mathcal{P}}_t$

First goal was to simply convert a vector u of length s to a vector v of length t. I did not succeed to do this. The code used can be found below. I have tried different sizes of s, t, values of u, p, α.

I compared $\frac{\tilde{\mathcal{A}}u}{\tilde{\mathcal{B}}\tilde{\mathcal{A}}u}$ to see if only a factor differed, this was not the case.

2.1 Global

```cpp
typedef long double poly_type;
typedef std::complex<poly_type> complex_type;
```

$|u_i| < 1$ for all elements of initial vector u to $\tilde{\mathcal{S}}'(u \in \mathbb{C}_{\circ})$

2.2 $\tilde{\mathcal{S}}'$ (page 27)

```cpp
std::vector<complex_type> gaussian_resampling_S(const std::vector<complex_type> &u,
        size_t s, size_t t, size_t a, size_t p) {
    size_t m = (size_t)std::ceil(std::sqrt((poly_type)p)) * a;
    std::vector<complex_type> tv(t, 0);
    for (size_t k = 0; k < t; k++) {
        // |j - (sk)/t| < m -> (-m + s*k/t, m + s*k/t)
        poly_type jstart = -(poly_type)m+((poly_type)s*(poly_type)k)/(poly_type)t + 1;
        poly_type jend   =  (poly_type)m+((poly_type)s*(poly_type)k)/(poly_type)t;
        for (poly_type j = jstart; j < jend; j++) {
            complex_type b = ((poly_type)1/(poly_type)2) * ((poly_type)1/(poly_type)a);

            complex_type x = -M_PI * std::pow((poly_type)a, -2);
            x *= std::pow((poly_type)j - ((poly_type)s * (poly_type)k)/(poly_type)t, 2);
            x = std::exp(x);

            complex_type y =  b * x;

            complex_type z = y * u[(size_t)(j + s) % s];
            tv[k] += z;
        }
    }
    return tv;
}
```

2.3 $\tilde{\mathcal{P}}_t$

Permutation, I have tried running this and then the inverse ($\tilde{\mathcal{P}}_t^{-1}$)

```cpp
std::vector<complex_type> gaussian_resampling_Pt(const std::vector<complex_type> &u,
        size_t s, size_t t) {
    std::vector<complex_type> ptv(t, 0);
    for (size_t k = 0; k < t; k++) {
        // +(poly_type)t*(poly_type)t), because s < t, k < t t*t will make it positive
        // ptv[k] = u[(size_t)((-(poly_type)s*(poly_type)k)+(poly_type)t*(poly_type)t) % t];
        // -sk mod t = (t-s)k mod t?
        ptv[k] = u[(t-s)*k % t];
    }
    return ptv;
}
```

2.4 $\tilde{\mathcal{C}}$

```cpp
std::vector<complex_type> gaussian_resampling_C(const std::vector<complex_type> &u,
        size_t s, size_t t) {

    std::vector<complex_type> csv(s, 0);
    for (size_t l = 0; l < s; l++) {
        csv[l] = u[(size_t)(std::round((poly_type)t*(poly_type)l)/(poly_type)s) % s];
    }
    return csv;
}
```

2.5 $\tilde{\mathcal{J}}'$

```cpp
std::vector<complex_type> gaussian_resampling_I(const std::vector<complex_type> &u,
        size_t s, size_t t, size_t a, size_t p) {
    std::vector<complex_type> isv(s, 0);
    std::vector<complex_type> visv(s, 0);

    auto ro = [](poly_type x) -> poly_type {
        return x >= 0 ? std::floor(x) : std::ceil(x);
    };
    auto c_ro = [ro](complex_type x) -> complex_type {
        return complex_type(ro(std::real(x)), ro(std::imag(x)));
    };

    // eps transformation
    auto eps = [](const std::vector<complex_type> &u, size_t s, size_t t, size_t a, size_t p) {
        auto beta = [t, s](size_t l) -> poly_type {
            return (poly_type)((poly_type)t*(poly_type)l)/(poly_type)s -
                std::round(((poly_type)t*(poly_type)l)/(poly_type)s);
        };

        poly_type m = std::ceil(std::sqrt(p)/(2*a));
        std::vector<complex_type> epssv(s, 0);
        for (size_t l = 0; l < s; l++) {
            for (poly_type h = -(poly_type)m; h <= (poly_type)m; h++) {
                if (h == 0) continue; // h not equal 0

                complex_type x = 1;
                x = -M_PI * std::pow(a, 2);
                x *= (std::pow((poly_type)t*h/(poly_type)s + beta(l), 2) -
                        std::pow(beta((size_t)(l+h+s) % s), 2));
                x = std::exp(x);

                complex_type z = x * u[(size_t)(l+h+s) % s];
                epssv[l] += z;
            }
        }
        return epssv;
    };

    // v = u/2 page: 30
    for (size_t i = 0; i < s; i++) {
        isv[i] = u[i]/(poly_type)2;
        visv[i] = u[i]/(poly_type)2;
        // complex_type tmp = ((poly_type)std::pow(2, -10)*c_ro((poly_type)std::pow(2,9)*u[i]));
        // std::cout << isv[i] << " ro: " << tmp << "\n";
    }

    poly_type n = std::ceil(p*s/(std::pow(a,2)*(t-s)));
    poly_type sign = 1;
    for (size_t i = 0; i < n; i++) { // i
        sign *= -1;
        visv = eps(visv, s, t, a, p);
        for (size_t i1 = 0; i1 < s; i1++) { // i
            isv[i1] += sign*visv[i1];
        }
    }

    return isv;
}
```

2.6 $\tilde{\mathcal{D}}'$

```
std::vector<complex_type> gaussian_resampling_D(const std::vector<complex_type> &u,
        size_t s, size_t t, size_t a) {
    auto beta = [t, s](size_t l) -> poly_type{
        return (poly_type)((poly_type)t*(poly_type)l)/(poly_type)s -
            std::round(((poly_type)t*(poly_type)l)/(poly_type)s);
    };

    std::vector<complex_type> dsv(s, 0);
    for (size_t l = 0; l < s; l++) {
        complex_type x = 1;
        x = M_PI * std::pow(a, 2) * std::pow(beta(l), 2);
        x = std::exp(x);
        x /= std::pow(2, 2 * std::pow(a, 2) - 2);
        dsv[l] = u[l]*x;
    }

    return dsv;
}
```

2.7 $\tilde{\mathcal{P}}_s^{-1}$

Permutation, I have tried running this and then the inverse ($\tilde{\mathcal{P}}_s$)

```
std::vector<complex_type> gaussian_resampling_Psinv(const std::vector<complex_type> &u,
        size_t s, size_t t) {
    std::vector<complex_type> pssv(s, 0);
    std::vector<std::tuple<size_t, complex_type>> tpssv(s);

    for (size_t i = 0; i < s; i++) {
        tpssv[i] = std::make_pair((t*i) % s, u[i]);
    }

    auto complex_cmp = [](const std::tuple<size_t, complex_type> &t1,
            const std::tuple<size_t, complex_type> &t2) {
        return (std::get<0>(t1) == std::get<0>(t2)
            && std::real(std::get<1>(t1)) < std::real(std::get<1>(t2)))
            || std::real(std::get<0>(t1)) < std::real(std::get<0>(t2));
    };

    std::sort(tpssv.begin(), tpssv.end(), complex_cmp);

    for (size_t i = 0; i < s; i++) {
        pssv[i] = std::get<1>(tpssv[i]);
    }

    return pssv;
}
```