Basic Polynomial Algebra Subprograms

Symbolic-Numeric Integration of Rational Functions with the BPAS Library

Marc Moreno Maza[1,2]

Joint work with

Robert M. Corless[2]   Robert H.C. Moir[2]   Ning Xie[2]

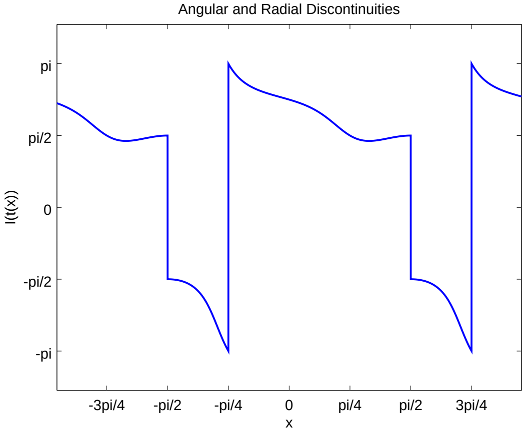
[11]CIGIT, Chinese Academy of Sciences, China
[2]University of Western Ontario, Canada

ChongQing Ocotober 10, 2015

- Suppose that, using the transformation $t = \tan(x)$, we obtain

$$I(t(x)) = \int \frac{2t^3-3t^2-1}{t^6+2t^3+t^2-2t+2}\,dt = \operatorname{arctan}\left(t^3+1,\, t-1\right).$$

It's discontinuous at $x = \frac{3\pi}{4} + n\pi$ and $x = \frac{\pi}{2} + n\pi, n \in \mathbb{Z}$.



Symbolic integration in BPAS

R. H. C. Moir, R. M. Corless, and D. J. Jeffrey (2014, July) present an algorithm

- where spurious discontinuities are handled via unwinding numbers (i.e. correcting terms)
- with an implementation in the BPAS library.

$$F(x) = \int f(x)\,dx.$$

For instance, it evaluates $\int \frac{x^4-3\,x^2+6}{x^6-5\,x^4+5\,x^2+4}\,dx = \mathrm{invtan}(x^3-3\,x, x^2-2)$.

```
  -
 /    6-3*x^2+1*x^4
 | -------------------- dx =
 / 4+5*x^2-5*x^4+1*x^6
-
     ----
     \     a*log((-2) + (6*a)*x + (1)*x^2 + (-2*a)*x^3)
     /
     ----
a|1/4+1*a^2=0
```

Contents

Plan

The full partial-fraction algorithm (1/4)

Notations

- Let $f \in \mathbb{R}(x)$ be a rational function over $\mathbb{R}$ not belonging to $\mathbb{R}[x]$.
- These exists $P, A, D \in \mathbb{R}[x]$ such that $f = P + A/D$ with $\gcd(A, D) = 1$ and $\deg(A) < \deg(D)$.
- Consider the irreducible factorization of D and write it:

$$D = c \prod_{i=1}^{n} (x - a_i)^{e_i} \prod_{j=1}^{m} (x^2 b_j x + c_j)^{f_j} \tag{1}$$

where $a_i, b_j, c_j \in \mathbb{R}$ and e_i, f_j are positive integers.

Full partial-fraction decomposition of f

There exist $A_{ik}, B_{jk}, C_{jk} \in \mathbb{R}$ such that we have

$$f = P + \sum_{i=1}^{n} \sum_{k=1}^{e_i} \frac{A_{ik}}{(x - a_i)^k} + \sum_{j=1}^{m} \sum_{k=1}^{f_j} \frac{B_{jk} x + C_{jk}}{(x^2 + b_j x + c_j)^k}. \tag{2}$$

The full partial-fraction algorithm (2/4)

Integration term by term

We have

$$\int f = \int P + \sum_{i=1}^{n}\sum_{k=1}^{e_i}\int \frac{A_{ik}}{(x-a_i)^k} + \sum_{j=1}^{m}\sum_{k=1}^{f_j}\int \frac{B_{jk}x + C_{jk}}{(x^2+b_jx+c_j)^k}, \qquad (3)$$

where

$$\int \frac{A_{ik}}{(x-a_i)^k} = \begin{cases} A_{ik}(x-a_i)^{k-1}/(k-1) & \text{if } k>1 \\ A_{i1}\log(x-a_i) & \text{if } k=1 \end{cases} \qquad (4)$$

and

$$\int \frac{B_{jk}x + C_{jk}}{x^2+b_jx+c_j} = \frac{B_{j1}}{2}\log(x^2+b_jx+c_j) + \frac{2C_{j1}-b_jB_{j1}}{\sqrt{4c_j-b_j^2}}\arctan\left(\frac{2x+b_j}{\sqrt{4c_j-b_j^2}}\right). \qquad (5)$$

with the case $k > 1$ handled recursively via integration by parts.

The full partial-fraction algorithm (3/4)

Alternative setting

- Consider an arbitrary field $\mathbb{K}$ of characteristic zero.
- Factor D over $\overline{\mathbb{K}}$, say $D = \prod_{i=1}^{q} (x - \alpha_i)^{e_i}$ and write

$$f = P + \sum_{i=1}^{q} \sum_{j=1}^{e_i} \frac{A_{ij}}{(x - \alpha_i)^j} \tag{6}$$

  which is equivalent to expanding f into its Laurent series at all its finite poles.
- Integrating the series term-wise, and then interpolating for the answer, by summing all the polar terms, leads to the following.

Liouville's theorem

There exist $v, u_1, \ldots, u_m \in \overline{\mathbb{K}}(x)$ and $c_1, \ldots, c_m \in \overline{\mathbb{K}}$ such that we have

$$\int f = v + c_1 \log(u_1) + \cdots + c_m \log(u_m) \tag{7}$$

The full partial-fraction algorithm (4/4)

Remarks

- Major computational inconvenience: factoring over $\mathbb{R}$, $\mathbb{C}$ or $\overline{\mathbb{K}}$ (and not just over $\mathbb{Q}$)
- Thereby introducing algebraic numbers even if the integrand and its integral are both in $\mathbb{Q}(x)$.
- Unfortunately, introducing algebraic numbers may be necessary: any field containing an integral of $1/(x^2+2)$ contains $\sqrt{2}$ as well.

Take away

Modern research has yielded so-called "rational" algorithms that

- compute as much of the integral as possible with all calculations being done in $\mathbb{K}(x)$, and
- compute the minimal algebraic extension of $\mathbb{K}$ necessary to express the integral.

Note: this section is based on the ISSAC 1998 tutorial of Manuel Bronstein.

The Hermite reduction (1/3)

Notations (using $\mathbb{K}$ instead of $\mathbb{R}$)

- Let $f \in \mathbb{K}(x)$ be a rational function over $\mathbb{K}$ not belonging to $\mathbb{K}[x]$.
- These exists $P, A, D \in \mathbb{K}[x]$ such that $f = P + A/D$ with $\gcd(A, D) = 1$ and $\deg(A) < \deg(D)$.
- Let $D = D_1 D_2^2 \cdots D_m^m$ be a square-free factorization of D.
- Thus the polynomials D_i are square-free and pairwise co-prime.
- Assume $m \geq 2$, define $V := D_m$ and $U := D/V^m$.

Objective

Reduce the integration of f to the case where D is square-free without introducing any algebraic numbers.

Calculations

1. Since $\gcd(UV', V) = 1$, we use the E. E. A. to compute $B, C \in \mathbb{K}[x]$ s.t.

$$\frac{A}{1-m} = BUV'' + CV \text{ and } deg(B) < \deg(V). \tag{8}$$

2. Multiplying both sides by $(1-m)/(UV^m)$ gives

$$\frac{A}{UV^m} = \frac{(1-m)BV'}{V^m} + \frac{(1-m)C}{UV^{m-1}}. \tag{9}$$

3. Adding and subtracting B'/V^{m--1} to the right hand side, we get

$$\frac{A}{UV^m} = \left(\frac{B'}{V^{m-1}} + \frac{(m-1)BV'}{V^m}\right) + \frac{(1-m)C - UB'}{UV^{m-1}}. \tag{10}$$

4. and integrating both sides yields

$$\int \frac{A}{UV^m} = \frac{B}{V^{m-1}} + \int \frac{(1-m)C - UB'}{UV^{m-1}}. \tag{11}$$

The Hermite reduction (3/3)

Recall

$$\int \frac{A}{UV^m} = \frac{B}{V^{m-1}} + \int \frac{(1-m)C - UB'}{UV^{m-1}}. \tag{12}$$

Hence, the integrand is reduced to one with a smaller power of V in the denominator. This process is repeated until the denominator is square-free, which leads to the following.

Hermite's theorem

Given $f \in \mathbb{K}(x)$, one can compute $g, h \in \mathbb{K}(x)$, such that

1. $f = g' + h$
2. the denominator of h is square-free.

The Lazard–Rioboo–Trager algorithm (1/3)

Setting

Following the Hermite reduction, we only have to integrate fractions of the form

$$f \quad = \quad A/D \quad \text{with} \quad \deg(A) < \deg(D) \quad \text{and} \quad D \quad \text{square-free.} \tag{13}$$

It follows from the full partial-fraction decomposition algorithm that

$$\int f \quad = \quad \sum_{i=1}^{i=n} a_i \log(x - \alpha_i) \tag{14}$$

where the α_i's are the zeros of D in $\overline{\mathbb{K}}$ and the a_i's are the residues of f at the α_i's.

Objective

The problem is then to compute those residues without splitting D into irreducible factors.

The Lazard–Rioboo–Trager algorithm (2/3)

Recall

Recall that we want

$$\int f = \sum_{i=1}^{i=n} a_i \log(x - \alpha_i) \tag{15}$$

Theorem (Rothstein and Trager (independently))

Let t be a new variable. Then, the a_i's are exactly the zeros of

$$R := \mathrm{resultant}(D, A - tD', x). \tag{16}$$

Moreover, the splitting field of R over $\mathbb{K}$ is the minimal algebraic extension of K necessary to express $\int f$ in the form given by Liouville's theorem and we have

$$\int \frac{A}{D} = \sum_{i=1}^{i=m} \sum_{a|R_i(a)=0} a \log(\gcd(D, A - aD')) \tag{17}$$

where $R = \prod_{i=1}^{i=m} R_i^{e_i}$ is the irreducible factorization of R over $\mathbb{K}$.

The Lazard–Rioboo–Trager algorithm (3/3)

Remark

The previous theorem still requires factoring R into irreducibles over $\mathbb{K}$, and computing GCDs over the algebraic extensions $\mathbb{K}[x]/\langle R_i(x)\rangle$.

Theorem (Trager and Lazard & Rioboo (independently))

Let $(R_0, R_1, \ldots, R_k, 0, 0, \ldots)$ be the sub-resultant chain of D and $A - tD'$ w.r.t. x. Let $R_o = Q_1 Q_2^2 \cdots Q_m^m$ be a square-free factorization of the resultant $R = R_0$. Then, we have

$$\int \frac{A}{D} = \sum_{i=1}^{i=m} \sum_{a|Q_i(a)=0} a \log(\gcd(D, A - aD')) \tag{18}$$

and the sum $\sum_{a|Q_i(a)=0} a \log(\gcd(D, A - aD'))$ is computed as follows

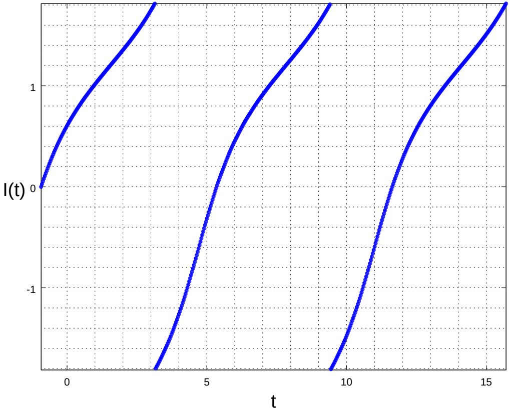
1. *if $i = \deg(D)$ then it is simply equal to $\sum_{a|Q_i(a)=0} a \log(D)$*
2. *otherwise, that is, if $i < \deg)D)$, denoting by R_{k_i} the sub-resultant of degree i (guaranteed to exist), the sum is equal to $\sum_{a|Q_i(a)=0} a \log(\mathrm{primitivePart}(R_{k_i}, x))$.*

Plan

- Many CAS return discontinuous expressions for continuous integrands, *e.g.*, in MAPLE

$$I(t) = \int \frac{dt}{2+\sin t} = \frac{2}{\sqrt{3}} \arctan\left(\frac{2\tan(\frac{t}{2})+1}{\sqrt{3}}\right),$$

discontinuous at $(2n+1)\pi, n \in \mathbb{Z}$.



Discontinuous anti-derivatives and Liouville's theorem

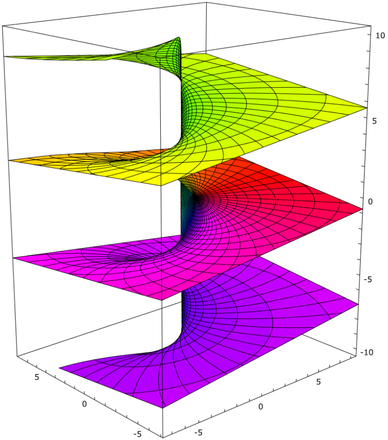
Liouville's theorem (recall)

For $f \in F$, where F is a differential field of characteristic 0 with $\mathrm{const}(F) = \mathbb{K}$, if there is a g in an elementary extension of F such that $\int f = g$, then there are $\varphi_i \in F$, $c_i \in \mathbb{K}$, such that $\int f = \varphi_0 + \sum_{i=1}^{n} c_i \log(\varphi_i)$.

Consequences

- if any of the functions φ_i have a complex value $\varphi_i = u + \imath v$, then the expression for the integral of f contains a term of the form
$$\log(\varphi) \quad = \quad \log(u + \imath v) \quad = \quad \log(\sqrt{u^2 + v^2}) \; + \; \imath \,\mathrm{arctangent}(\tfrac{v}{u}).$$
- If u has a root or v has a pole on the path of integration, then the integral of f will have a spurious discontinuity.
- Because $\frac{-v}{u} = \frac{v}{-u}$, there is also a spurious discontinuity at $\pm\frac{\pi}{2}$.
- If φ crosses the branch cut for the logarithm along the path of integration, then an additional spurious discontinuity arises from the choice of logarithm function.

The complex logarithm



A plot of the multi-valued imaginary part of the complex logarithm function, which shows the branches. As a complex number z goes around the origin, the imaginary part of the logarithm goes up or down. This makes the origin a branch point of the function.

Spurious discontinuities

Spurious discontinuities

Spurious discontinuities arising from

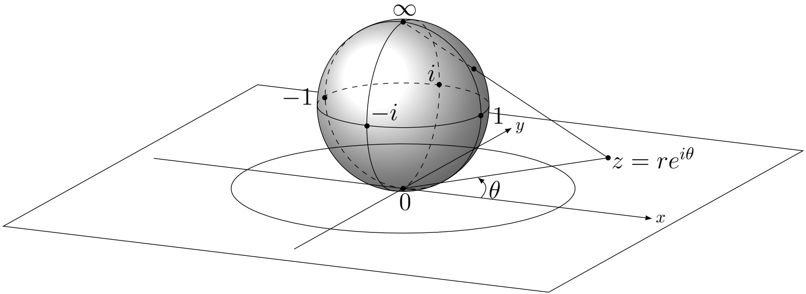
- choosing a branch of the logarithm, or
- singular arguments of the arctangent function

are a quite generic feature of integrals as a result of Liouville's theorem.

Dealing with spurious discontinuities

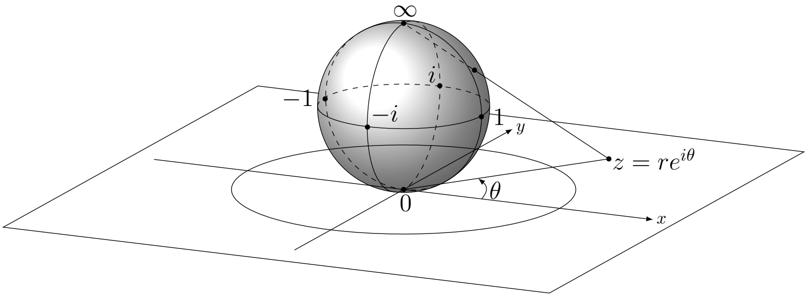
- The winding number solution that Corless and Jeffrey propose provides a solution to the problem of paths of integration that implicitly cross branch cuts of complex functions
- The continuity issue arising in cases like that from a diverging argument to an arctangent function was first studied by Jeffrey and further improved by Corless, Jeffrey and Moir. We shall expand on that second kind of discontinuity.
- Along the path of integration, the values of arguments of functions in the expression of the integral can pass through the point at infinity of the Riemann sphere and safely emerge from it without leading to dis- continuity.
- The tangent function and its inverse form a natural example.

Unwinding paths on the Riemann sphere (1/3)



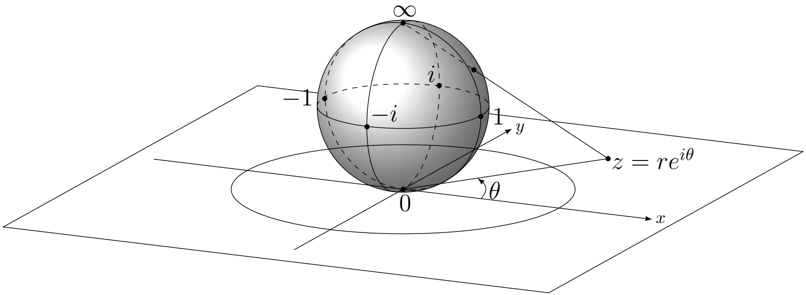
- The image of the real line on the Riemann sphere is a great circle if we include the point at infinity.
- We will call $\mathbb{R} \cup \{\infty\}$ on the Riemann sphere the *real circle*.
- Consider the continuous behavior of r on the real circle.
- If we increase r from 0 to π, then $\tan(r)$ completes a full loop around the real circle, passing through ∞ at $r = \pi/2$.

Unwinding paths on the Riemann sphere (2/3)



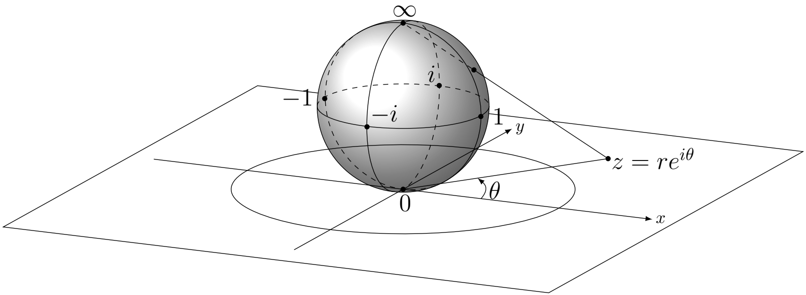
- Turning things around: if we allow r to complete a full loop counterclockwise around the real circle starting from 0 (0 to $+\infty$) and back to 0 from $-\infty$ on the real line), then $\arctan(r)$ increases from 0 to π, taking the value $\frac{\pi}{2}$ at $r = \infty$.
- In this way, by keeping track of windings around the real circle passing through the point at infinity, we can define the tangent function over the entire real line and the arctangent function such that $\arctan(\tan r) = r$ for all r on the entire real line.

Unwinding paths on the Riemann sphere (3/3)



In summary, unwinding paths on the Riemann sphere handle two sources of discontinuity:

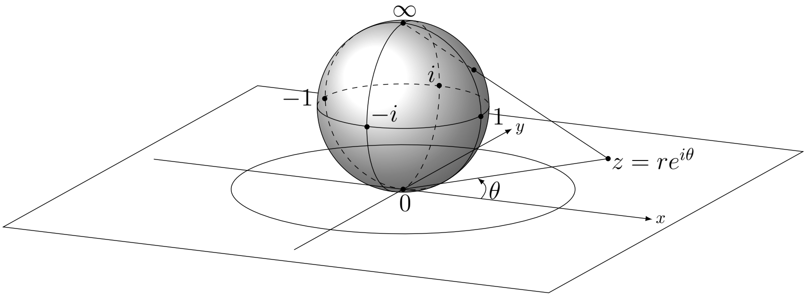
Unwinding paths on the Riemann sphere (3/3)



In summary, unwinding paths on the Riemann sphere handle two sources of discontinuity:

- logarithmic branch cut crossings (change θ by $\pm 2\pi$).

Unwinding paths on the Riemann sphere (3/3)



In summary, unwinding paths on the Riemann sphere handle two sources of discontinuity:

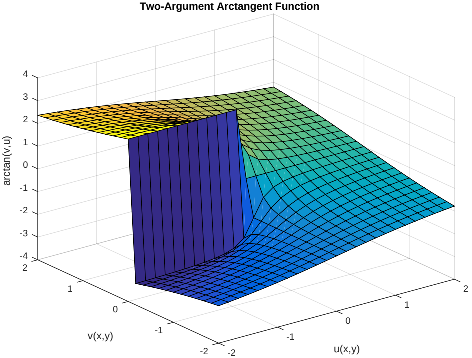
- logarithmic branch cut crossings (change θ by $\pm 2\pi$).
- pole-type passes through ∞ (change θ by $\pm\pi$).

Two-argument arctangent function

Definition

If $z = x + iy = re^{i\theta}$, then $\arctan(y, x) := \theta$.

- This notion is necessary to preserve quadrant information, which the one-argument arctangent destroys because $(-y)/x = y/(-x)$.
- This automatically removes spurious discontinuities from poles of rational functions.



Unwinding numbers

Angular unwinding number

- The *unwinding number* $\mathcal{K}(z)$ for the complex logarithm is defined as follows:
- For a complex number $z = x + iy$, we write $log(e^z) = z - 2\pi i\mathcal{K}(z)$ or without reference to logarithms, $\mathcal{K}(z) = \mathcal{K}(iy) = \left\lceil \frac{y-\pi}{2\pi} \right\rceil$.

Radial unwinding number

To fully "unwind" the arctangent we require two unwinding numbers, one for regular and singular multivaluedness:

$$arctan(v,u) := Arctan(v,u) + 2\pi\mathcal{K}_\theta(v,u) + \pi\mathcal{K}_r(v,u).$$

- regular multivaluedness: *angular unwinding number* $\mathcal{K}_\theta(v,u)$ for logarithmic branch cut crossings (± 1 for counterclockwise/clockwise crossings);
- singular multivaluedness: *radial unwinding number* $\mathcal{K}_r(v,u)$ for poles in v of odd order ($\mathcal{K}_r = \pm 1$ for decreasing/increasing θ).

Unwinding numbers and Liouville's theorem

Theorem

Let $a, b \in \mathbb{R}$ with $a < b$. Let $p, q \in \mathbb{R}[x]$ with $\gcd(p, q) = 1$, $\deg(p) < \deg(q)$ and $q(x) \neq 0$ on the interval $[a, b]$. Then, there exist computable functions, called *unwinding numbers* $\mathcal{K}_r, \mathcal{K}_\theta : [a, b] \to \mathbb{Z}$ such that $x \longmapsto \int_a^x \frac{p(t)}{q(t)} dt$ is continuous on $[a, b]$ and can be constructed either in terms of the two-argument arctangent function as

$$\begin{aligned} \int_a^x \frac{p(t)}{q(t)} dt \quad = \quad & a_0 \frac{p_0(x)}{q_0(x)} \; + \; \sum_{k=1}^m a_k \log(p_k(x)) \; + \\ & \sum_{k=m+1}^n a_k \operatorname{arctang}(p_k(x), q_k(x)) \; + \; 2\pi \mathcal{K}_\theta(x), \end{aligned} \tag{19}$$

or in terms of the one-argument arctangent function and $\mathcal{K}_r$,.

Plan

The algorithm

- input:
  - $f(x) = p(x)/q(x)$, $\deg(q) > \deg(p)$
  - error tolerance $\varepsilon > 0$

The algorithm

- input:
  - $f(x) = p(x)/q(x)$, $\deg(q) > \deg(p)$
  - error tolerance $\varepsilon > 0$
- output $\int \hat{f}\, dx = p_0/q_0 + \sum c_i \log(p_i) + \sum c_j \mathrm{arctan}(p_j, q_j)$, such that the coefficients of $\hat{f}$ differ from those of f by no more than ε.

The algorithm

- input:
  - $f(x) = p(x)/q(x)$, $\deg(q) > \deg(p)$
  - error tolerance $\varepsilon > 0$
- output $\int \hat{f}\, dx = p_0/q_0 + \sum c_i \log(p_i) + \sum c_j \arctan(p_j, q_j)$, such that the coefficients of $\hat{f}$ differ from those of f by no more than ε.
- main steps:
  - decompose into $\int f\, dx = \frac{p_0}{q_0} + \int \frac{p_r}{q_r}\, dx$ using Hermite reduction.
  - compute transcendental part $\displaystyle\int \frac{p_r}{q_r}\, dx = \sum_i \sum_{\mathrm{rootsOf}(U_i(t))} t \cdot \log(S_i(t, x))$ symbolically using Lazard–Rioboo–Trager algorithm.
  - compute roots of $U_i(t)$ numerically using MPSolve to precision ε/γ (γ defined below).
  - symbolic post-processing in BPAS: compute log and arctan terms.

Comparison With Prior Approaches

A number of symbolic-numeric integration algorithms have been proposed in the literature:

Comparison With Prior Approaches

A number of symbolic-numeric integration algorithms have been proposed in the literature:

- Notable examples are (Noda 2007), (Fateman 2008)that have numerical computations based on numerical rootfinding and use hybrid approximate GCD algorithms, which handle noisy input data.

Comparison With Prior Approaches

A number of symbolic-numeric integration algorithms have been proposed in the literature:

- Notable examples are (Noda 2007), (Fateman 2008)that have numerical computations based on numerical rootfinding and use hybrid approximate GCD algorithms, which handle noisy input data.
- These methods vary in the symbolic integration method, but use partial fraction decomposition (PFD) with or without the Rothstein-Trager (RT) algorithm.

Our approach differs in several ways:

- uses the Lazard-Rioboo-Trager algorithm (more efficient than RT and full PFD) with modular gcd and subresultant computations for speed;
- uses the highly optimized multiprecision numerical rootfinding package MPSolve to allow tolerance controlled symbolic integration;
- symbolic algorithms are implemented in BPAS (basic polynomial algebra subprograms).
  - written in C++, it is a highly optimized system for symbolic computations on (multi-core) parallel architectures.

Note: our approach can be extended to incorporate approximate gcd computations.

Plan

Theorem: Backward Stability of the Algorithm

Given a rational function $f = p/q$ and tolerance ε, the above algorithm yields an integral of a rational function $\hat{f}$ with coefficients differing from those of f by less than ε.

Sketch of Proof:

- Assume that we have the exact integral in the form

$$\int f\,dx = \sum_i \sum_{\text{rootsOf}(U_i(t))} t \cdot \log(S_i(t,x))$$

  that $U_i(t)$ has degree r, and that each root c_j of $U_i(t)$ has relative error $\delta_j \leq \mu_w = 2^{-w}$.
- Consider a single term of the form $\sum_{\text{rootsOf}(U(t))} t \cdot \log(S(t,x))$ and assume that it differentiates to p/q with exact roots.

Sketch of Proof (cont'd):

- Let c_j be the exact roots, then with substitution and differentiation we have
$$\hat{f} = \sum_j c_j(1+\delta_j)\frac{S'(c_j(1+\delta_j),x)}{S(c_j(1+\delta_j),x)},$$
with $\hat{f} \to f$ as $\delta \to 0$ for $\delta = \max_j \delta_j$.
- Now, the product $\prod_j S(c_j, x) = q(x)$, so, the largest error factor multiplying any of the coefficents of $\prod_i S(c_j(1+\delta_j), x)$, the denominator of $\hat{f}$, is bounded by $(1+\delta)^m = (1+m\delta) + O(\delta^2)$ for $m = r \cdot \deg_t(S(t,x))$.
- Then, $\sum_j c_j S'(c_j, x)(\prod_{k,k\neq i} S(c_k, x)) = p(x)$, from which it follows that the largest error factor in coefficients of the numerator of $\hat{f}$ is $(1+\delta)^\ell = (1+\ell\delta) + O(\delta^2)$ for $\ell = (r-1)\cdot\deg_t(S(t,x)) + \deg_t(S'(t,x)) + 1$.

Sketch of Proof (cont'd):

- Let us re-introduce the index i for each symbolic sum, giving the error exponents ℓ_i and m_i for each term.
- Arguing similarly to the single term case, we find that the error exponent for $q(x)$ is $\beta := \sum_i m_i$ and for $p(x)$ it is $\alpha := \max_i \alpha_i$, where

$$\alpha_i = \ell_i + \sum_{k, k \neq i} m_k$$

- The maximum error exponent for any coefficient of $\hat{f}$ is then

$$\gamma := \max\{\alpha, \beta\}.$$

- A valid tolerance for the root computation is therefore selecting the tolerance ε/γ for the numerical root computations.

Plan

Basic Polynomial Algebra Subprograms

MPSolve

Our implementation uses

- symbolic computation in BPAS (`bpaslib.org`)
  - polynomial arithmetic operations (multiplication, division, root isolation, etc.) for univariate and multivariate polynomials over prime fields or with integer or rational number coefficients.
- numerical computation in MPSOLVE (`numpi.dm.unipi.it/mpsolve`)
  - arbitrary precision solver for polynomials and secular equations, guaranteed inclusion radii, even on restricted domains.

Output formats

```
 -
/   1
| ----- dy =
/ 1+y^4
-
-0.176777*ln(abs(2-2.82843*y+2*y^2)) + 0.176777*ln(abs(2+2.82843*y+2*y^2))
- 0.353553*arctan(1-1.41421*y,1) - 0.353553*arctan(-1-1.41421*y,1)
```

The implementation of symbolic integration currently features:

- output in approximate floating point or rational number formats;
- output formatting for MAPLE and MATLAB.

- To illustrate the performance of the code, consider the integral

$$\int \frac{dx}{1+y^4},$$

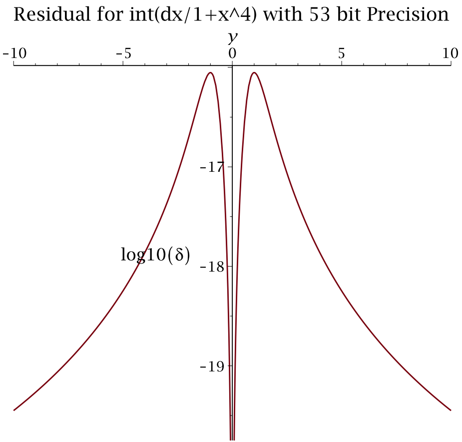
for which MAPLE provides the expression

$$(1/4)\sqrt{2}\arctan(y\sqrt{2}+1)+(1/4)\sqrt{2}\arctan(y\sqrt{2}-1)+(1/8)\sqrt{2}\ln((y^2+y\sqrt{2}+1)/(y^2-y\sqrt{2}+1)).$$

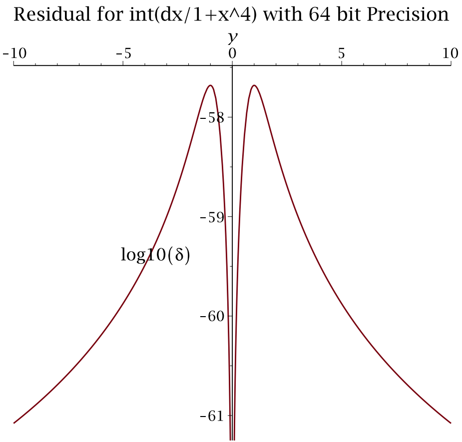
- In floating point display format, BPAS returns the output shown on the previous slide:

$$-0.0243257*ln(abs(2.83849-2.58171*y+2.32183*y^2))+$$
$$0.111397*ln(abs(8.94592-2.16075*y+2.29985*y^2))+$$
$$0.825858*ln(abs(1.05145+y))+0.136421*arctan(-0.581714+1.04632*y,1)+$$
$$0.177316*arctan(0.245242-0.522059*y,1)$$

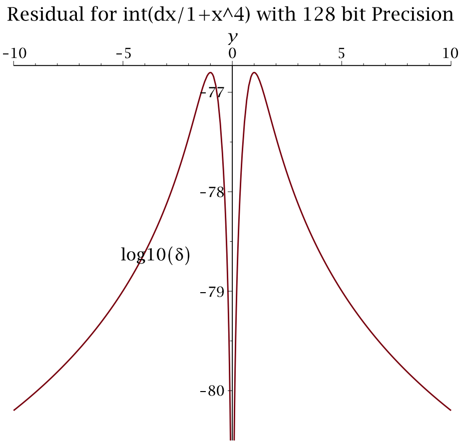
- By differentiating the expression returned by intRF, we can compute a residual,



- By differentiating the expression returned by intRF, we can compute a residual, which exhibits a kind of tolerance proportionality:



- By differentiating the expression returned by intRF, we can compute a residual, which exhibits a kind of tolerance proportionality:



- [?] compares his algorithm to a number of numerical integration methods on the following three definite integration problems:
  1. A singularity outside the integral region but close to both ends:
  $$I_1 = \int_0^1 \frac{dx}{1000x(x-1) - 0.001}.$$
  2. The integrand has a sharp peak in the integral region:
  $$I_2 = \int_0^1 \frac{dx}{1000(x-0.5)^2 + 0.001}.$$
  3. The integrand has both the above two properties:
  $$I_3 = \int_0^1 \frac{dx}{x^5 - x^4 - 0.75x^3 + x^2 - 0.25x - 10^{-6}}.$$

- Noda points out that the hybrid integration algorithm described performs well in comparison to Gaussian quadrature of 32 points (G32), double-exponential formula (DE), the Romberg method (Romb) and the adaptable Newton-Cotes method (NC), as shown in the following table:

	symbolic-numeric	numeric			
	hybrid	G32	DE	Romb	NC
I1	-0.02763097	-0.01622972	-0.02763097	-0.68482819	5.40966656
I2	3.13759266	0.19994034	24.6736125	2.98225113	3.13759258
I3	-5195.24497	-580.39410	-24965.007	-5759.10716	230.75544

- This looks a little less impressive when we compare this with the `integral` command in MATLAB:

	symbolic-numeric	numeric
	hybrid	`integral` (MATLAB)
I1	-0.02763097	-0.027630969854033
I2	3.13759266	3.137592658923841
I3	-5195.24497	-5195.244973459549

- We consider our code, implemented as the C++ function `intRF`, by comparing to `integral` and the `int` command of MAPLE on the three problems, we find the following results:

symbolic	symbolic-numeric		numeric
`int` (MAPLE)	`intRF` (BPAS)	hybrid	`integral` (MATLAB)
-0.02763097	-0.02763097 (10^{-18})	-0.02763097	-0.02763097 (10^{-13})
3.13759266	3.13759266 (0)	3.13759266	3.13759266 (10^{-13})
-5195.24497	-5195.24497 (10^{-38})	-5195.24497	-5195.24497 (10^{-12})

- To see that our code is performing the way that we want, we can focus on problem I1 with a variable parameter ϵ

$$I_1(\epsilon) = \int_0^1 \frac{dx}{1000x(x-1) - \epsilon}$$

and decrease ϵ, which yields the following results

	symbolic	symbolic-numeric	numeric
ϵ	`int` (MAPLE)	`intRF` (BPAS)	`integral` (MATLAB)
10^{-3}	-0.02763097	-0.02763097 (10^{-18})	-0.02763097 (10^{-13})
10^{-6}	-0.041446531	-0.041446531 (10^{-14})	-0.041446531 (10^{-10})
10^{-10}	-0.059867212	-0.059867212 (10^{-14})	-0.059877278 (10^{-4})
10^{-14}	-0.078287893	∞	-0.074122442 (10^{-2})

- To see that our code is performing the way that we want, we can focus on problem I1 with a variable parameter ϵ

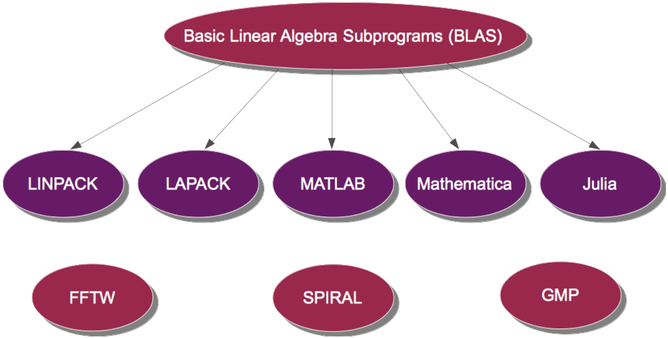
$$I_1(\epsilon) = \int_0^1 \frac{dx}{1000x(x-1)-\epsilon}$$

and decrease ϵ, which yields the following results

	symbolic	symbolic-numeric	numeric
ϵ	`int` (MAPLE)	`intRF` (BPAS)	`integral` (MATLAB)
10^{-3}	-0.02763097	-0.02763097 (10^{-18})	-0.02763097 (10^{-13})
10^{-6}	-0.041446531	-0.041446531 (10^{-14})	-0.041446531 (10^{-10})
10^{-10}	-0.059867212	-0.059867212 (10^{-14})	-0.059877278 (10^{-4})
10^{-14}	-0.078287893	-0.078287893 (10^{-35})	-0.074122442 (10^{-2})

Plan

Building blocks in scientific software



- No symbolic computation software dedicated to *sequential polynomial arithmetic* managed to play the unification role that the BLAS play in numerical linear algebra.
- Could this work in the case of hardware accelerators?
- How to benefit from other successful projects related to polynomial arithmetic, like `FFTW`, `SPIRAL` and `GMP`?

The *Basic Polynomial Algebra Subprograms*

Driving observation

▹ Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation.

▹ Algebraic complexity is often estimated in terms of multiplication time. At the software level, this reduction to multiplication is also common (Magma, NTL, FLINT, . . .).

▹ BPAS design follows the principle *reducing everything to multiplication*.

Targeted functionalities

Level 1: core routines specific to a coefficient ring or a polynomial representation: multi-dimensional FFTs, SLP operations, . . .

Level 2: basic arithmetic operations for dense or sparse polynomials with coefficients in $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{Z}/p\mathbb{Z}$: polynomial multiplication, Taylor shift, . . .

Level 3: advanced arithmetic operations taking as input a zero-dimensional regular chains: normal form of a polynomial, multivariate real root isolation, . . .

Targeted architectures

- The BPAS library http://www.bpaslib.org is written in C++ with CilkPlus http://www.cilkplus.org/ extension targeting multi-cores.
- Programs on multi-core processors can be written in CilkPlus or OpenMP. Our Meta_Fork framework http://www.metafork.org performs automatic translation between the two as well as conversions to C/C++.
- Graphics Processing Units (GPUs) with code written in CUDA, provided by the CUMODP library http://www.cumodp.org.
- Unifying code for both multi-core processors and GPUs is conceivable (see the SPIRAL project) but highly complex (multi-core processors enforce memory consistency while GPUs do not, etc.)

Implementation techniques

Level 1: core routines

- code is highly optimized in terms of work, data locality and parallelism,
- automatic code generation is used at library installation time.

Level 2: basic arithmetic operations

- functions provide a variety of algorithmic solutions for a given operation,
- the user can choose between algorithms minimizing work or algorithms maximizing parallelism.
- Example: Schönaghe-Strassen, divide-and-conquer, k-way Toom-Cook and the two-convolution method for integer polynomial multiplication.

Level 3: advanced arithmetic operations

- functions combine several Level 2 algorithms for achieving a given task,
- this leads to adaptive algorithms that select appropriate Level 2 functions depending on available resources (number of cores, input data size).

Plan

Code organization

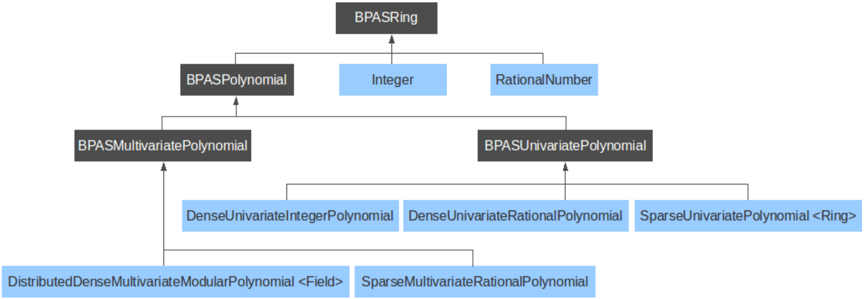
Subprojects

- Polynomial types with specified coefficient ring: `ModularPolynomial/`, `IntegerPolynomial/` and `RationalNumberPolynomial/`.
- Polynomial types with unspecified coefficient ring (template classes): `Polynomial/`.
- `ModularPolynomial/` is based on the Modpn library and includes our FFT code generator, which is inspired by FFTW and SPIRAL.
- `IntegerPolynomial/` relies on the GMP library.

User interface

- The UI currently exposes part of the polynomial types (the univariate ones and sparse multivariate polynomials)
- Exposing the other ones is work in progress.
- But the entire project is freely available in source at www.bpaslib.org.

User interface



- The above is a snapshot of the BPAS ring classes
- This shows two multivariate polynomial concrete classes, namely `DistributedDenseMultivariateModularPolynomial<Field>` and `SMQP`, and three univariate polynomial ones, namely `DUZP`, `DUQP` and `SparseUnivariatePolynomial<Ring>`.
- The BPAS classes `Integer` and `RationalNumber` are BPAS wrappers for GMP's `mpz` and `mpq` classes.
- Many other classes are provided like `Intervals`, `RegularChains`, ...

User interface: code example

```
#include <bpas.h>

int main (int argc, char *argv[] ) {
    DUZP a (4096), b (4096); // Initializing space
    for (int i = 0; i < 4096; ++i) { a.setCoefficient(i, rand()%1000+1); }
    for (int i = 0; i < 4096; ++i) { b.setCoefficient(i, rand()%1000+1); }
    DUZP c = (a^2) - (b^2), d = (a^3) - (b^3);
    DUZP g = c.gcd(d); // Gcd computation, g = a - b
    c /= g; // Exact division, c = a + b
    std::cout << "g = " << g << std::endl;

    DUQP p; // Initializing as a zero polyomial
    p = (p + mpq_class(1) << 4095) + mpq_class(4095); // p = x^4095 + 4095
    Intervals boxes = p.realRootIsolation(0.5);
    std::cout << "boxes = " << boxes << std::endl;

    SMQP f(3), g(2); // Initializing with number of variables
    SMQP h = (f^2) + f * g * mpq_class(2) + (g^2);
    SparseUnivariatePolynomial<SMQP> s = h.convertToSUP("x");
    SMQP z (s);
    if (z != h) { std::cout << z << " & " << h << " should not differ " << std::endl; }

    return 0;
}
```

Plan

Three core subprograms

- One-dimensional modular FFTs
- Parallel dense integer polynomial multiplication
- Parallel Taylor shift computation $f(x) \longmapsto f(x+1)$

Discrete Fourier Transform (DFT)

Definition

Given a primitive n-th root of unity ω (i.e. $\omega^{n/2} = -1$), and

$$f(t) = x_0 + x_1 t + \cdots + x_{n-1} t^{n-1},$$

$\mathrm{DFT}_n^{\omega}(f)$ is $\mathbf{y} = (y_0, \ldots, y_{n-1})$ with $y_k = f(\omega^k)$ for $0 \leq k < n$. As a matrix-vector product, it is

$$\mathbf{y} = \mathrm{DFT}_n \mathbf{x}, \quad \mathrm{DFT}_n = [\omega^{k\ell}]_{0 \leq k,\, \ell < n}. \tag{20}$$

Example

$$\begin{cases} y_0 = x_0 + x_1 \\ y_1 = x_0 - x_1 \end{cases} \iff \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

That is, $\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

Cooley-Tukey FFT

Cooley-Tukey FFT: factorization formula

$$\mathrm{DFT}_{qs} = (\mathrm{DFT}_q \otimes I_s) D_{q,s} (I_q \otimes \mathrm{DFT}_s) L_q^{qs},$$

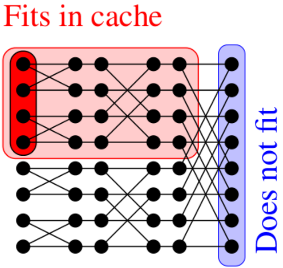
Cooley-Tukey FFT: iterative formula

$$\mathrm{DFT}_{2^k} = \left(\prod_{i=1}^{k} (I_{2^{i-1}} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i}}) \, T_{n,i} \right) R_n$$

with the twiddle factor matrix $T_{n,i} = I_{2^{i-1}} \otimes D_{2,2^{k-i}}$ and the bit-reversal permutation matrix

$$R_n = (I_{n/2} \otimes L_2^2)(I_{n/2^2} \otimes L_2^4) \cdots (I_1 \otimes L_2^n).$$

1-D FFTs: classical cache friendly algorithm



FFT($[a_0, a_1, \cdots, a_{n-1}], \omega$)

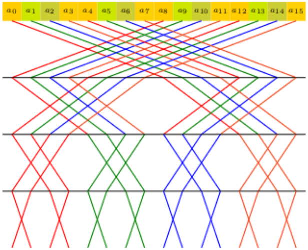
---

if $n \leq$ HTHRESHOLD then
  ArrayBitReversal($a_0, a_1, \cdots, a_{n-1}$)
  return FFT_iterative_in_cache($[a_0, a_1, \cdots, a_{n-1}], \omega$)
end if
Shuffle($a_0, a_1, \cdots, a_{n-1}$)
$[a_0, a_1, \cdots, a_{n/2-1}]$=FFT($[a_0, a_1, \cdots, a_{n/2-1}], \omega^2$)
$[a_{n/2}, a_{n/2+1}, \cdots, a_{n-1}]$=FFT($[a_{n/2}, a_{n/2+1}, \cdots, a_{n-1}], \omega^2$)
return $[a_0 + a_{n/2}, a_1 + \omega \cdot a_{n/2+1}, \cdots, a_{n/2-1} - \omega^{n/2-1} \cdot a_{n-1}]$

Cache friendly 1-D FFT

- If the input vector does not fit in cache, a recursive algorithm is applied
- Once the vector fits in cache, an iterative algorithm (not requiring shuffling) takes over.
- On an ideal cache of Z words with L words per cache line this yields a cache complexity of $\Omega(n/L(\log_2(n) - \log_2(Z)))$ which is not optimal.

1-D FFTs: cache complexity optimal algorithm



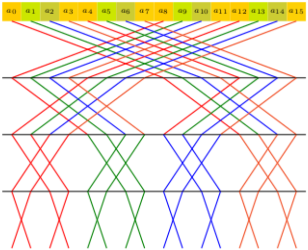
Fast Fourier Transform in R

procedure $\mathrm{FFT}((\alpha_0\alpha_1 ... \alpha_{N-1}), \omega, N = J \cdot K, \Omega = \omega^{N/K})$
  for $0 \leq k < K-1$ **do** ▷ Inner transforms
    for $0 \leq k' < J-1$ **do**
      $\gamma[k][k'] = \alpha_{k'K+k}$
    end for
    $c[k] = FFT(\gamma[k], \omega^K, J, \Omega)$
  end for
  for $0 \leq j < J-1$ **do** ▷ Outer transforms
    for $0 \leq k < K-1$ **do**
      $\delta[j][k] = c[k][j] * \omega^{jk}$ ▷ Computation of coefficients
    end for
    $d[j] = FFT(\delta[j], \omega^J, K, \Omega)$
    for $0 \leq j' < K-1$ **do**
      $\beta_{j'J+j} = d[j][j']$
    end for
  end for
  return $b = (\beta_0, ..., \beta_{N-1})$
end procedure

Cache optimal 1-D FFT

- Instead of processing row-by-row, one computes as deep as possible while staying in cache (resp. registers): this yields a blocking strategy.
- On the left picture, assuming $Z = 4$, on the first (resp, last) two rows, we successively compute the red, green, blue, orange 4-point blocks.
- On an ideal cache of Z words with L words per cache line the cache complexity drops to $\boxed{O(n/L(\log_2(n)/\log_2(Z))) \text{ which is optimal.}}$

1-D FFTs in BPAS: putting Fürer's algorithm into practice



Second Strategy FFT

procedure FFT$((\alpha_0\alpha_1\ldots\alpha_{N-1}),\ \omega,\ N = 2^{k-j}\cdot 2^j,\ \Omega = \omega^{2^{k-j}})$
  for $0 \le l < 2^j - 1$ **do** ▷ Inner transforms
    for $0 \le l' < 2^{k-j} - 1$ **do**
      $\gamma[l][l'] = \alpha_{l'2^j+l}$
    end for
    $c[l] = FFT(\gamma[l], \omega^K, 2^{k-j}, \Omega)$
  end for
  for $0 \le i < 2^{k-j} - 1$ **do** ▷ Outer transforms
    for $0 \le l < 2^j - 1$ **do**
      $\delta[i][l] = c[l][i] * \omega^{il}$ ▷ Computation of coefficients
    end for
    $d[i] = FFT(\delta[i], \omega^{2^{k-j}}, 2^j, \Omega)$
    for $0 \le i' < 2^j - 1$ **do**
      $\beta_{i'2^{k-j}+i} = d[i][i']$
    end for
  end for
  return $b = (\beta_0, \ldots, \beta_{N-1})$
end procedure

Cache-and-work optimal 1-D FFT

- Modifying the previous blocking strategy such that each block is an FFT on 2^K points, for a given K (small in practice), and
- choosing a sparse radix prime p (like $p = r^4 + 1$, for $r = 2^{16} - 2^8$) such that multiplying by the twiddle factors is cheap enough,
- the algebraic complexity drops from $O(n \log_2(n))$ to $\boxed{O(n \log_K(n))\ \text{which is optimal on today's desktop computers.}}$

1-D FFTs in BPAS

- In addition to the above optimal blocking strategy, instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used (SSE2, SSE4) and instruction pipeline usage is highly optimized.
- BPAS 1-D FFT code automatically generated by configurable Python scripts.

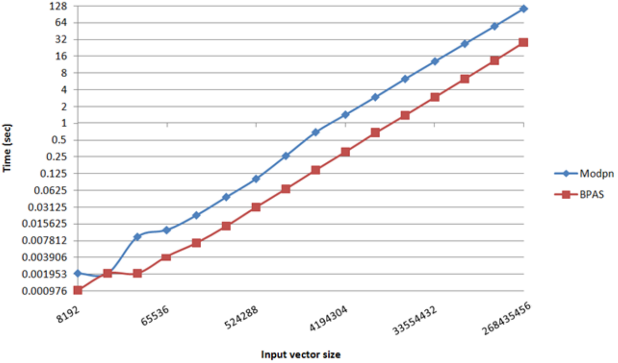
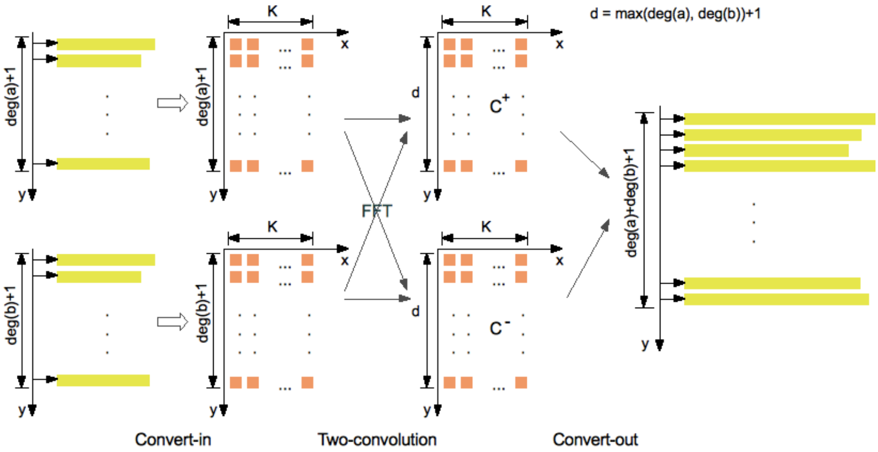


Figure: 1-D modular FFTs: Modpn (**serial**) vs BPAS (**serial**).

Parallel dense integer polynomial multiplication

A new algorithm: the two-convolution method



- work is $O(s \log_K(s))$, where s is the maximum bit-size of an input;
- parallelism is $O(\frac{\sqrt{s}}{\log_2(s)})$.

Parallel dense integer polynomial multiplication

The adaptive algorithm based on the input size and available resources

- Very small: Plain multiplication
- Small or Single-core: KS+SS algorithm
- Big but a few cores: 4-way Toom-Cook
- Big: 8-way Toom-Cook
- Very big: Two-convolution method

Parallel dense integer polynomial multiplication

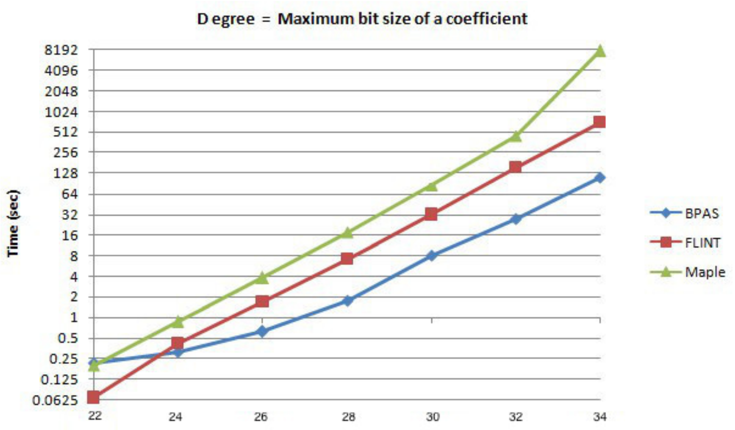


Figure: BPAS (parallel) vs FLINT (serial) vs Maple 18 (serial) with the logarithmic scale in radix 2 of the maximum bit-size of an input polynomial as the horizontal axis.

Parallel dense integer polynomial multiplication

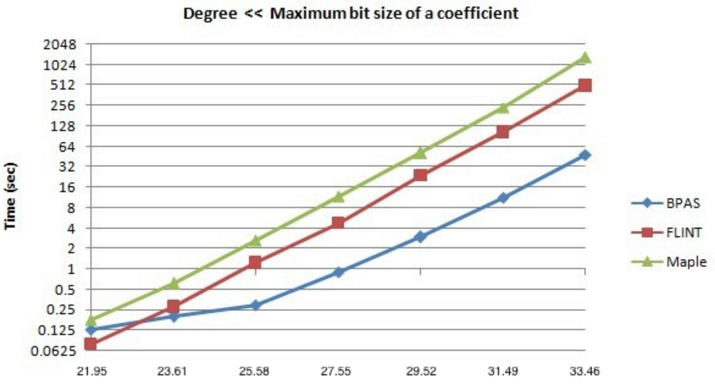


Figure: BPAS (parallel) vs FLINT (serial) vs Maple18 (serial) with the logarithmic scale in radix 2 of the maximum bit-size of an input polynomial as the horizontal axis.

Plan

Applications

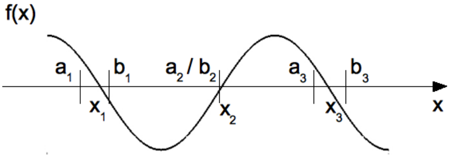
- Parallel univariate real root isolation
- Parallel multivariate real root isolation
- Symbolic integration

Parallel univariate real root isolation

Input: A univariate squarefree polynomial $f(x) = c_d x^d + \cdots + c_1 x + c_0$ with rational number coefficients

Output: A list of pairwise disjoint intervals $[a_1, b_1], \ldots, [a_e, b_e]$ with rational endpoints such that

- each real root of $f(x)$ is contained in one and only one $[a_i, b_i]$;
- if $a_i = b_i$, the real root $x_i = a_i(b_i)$; otherwise, the real root $a_i < x_i < b_i$ ($f(x)$ doesn't vanish at either endpoint).



Parallel univariate real root isolation

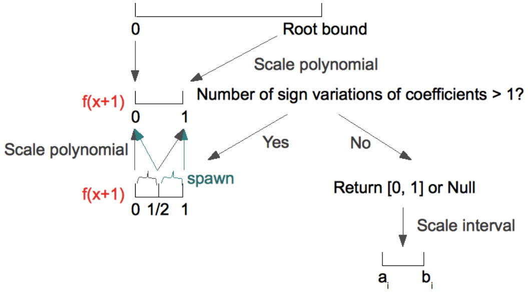


Figure: Parallel Vincent-Collins-Akritas (VCA, 1976)

- The most costly operation is the Taylor Shift operation, that is, the map $f(x) \longmapsto f(x+1)$.

Parallel univariate real root isolation

We run two parallel real root algorithms, BPAS and CMY [3], which are both implemented in CilkPlus, against Maple 18 serial *realroot* command (interface of the RUR-based code implemented in C by F. Rouillier) which implements a state-of-the-art algorithm.

	Size	BPAS (Parallel)	CMY [3] (Parallel)	*realroot* (Serial)	$\frac{T_{\text{CMY}}}{T_{\text{BPAS}}}$	$\frac{T_{realroot}}{T_{\text{BPAS}}}$	#Roots
Cnd	32768	18.141	125.902	816.134	6.94	44.99	1
	65536	66.436	664.438	7,526.428	10.0	113.29	1
Chebycheff	2048	608.738	594.82	1,378.444	0.98	2.26	2047
	4096	8,194.06	10,014	35,880.069	1.22	4.38	4095
Laguerre	2048	1,336.14	1,324.33	3,706.749	0.99	2.77	2047
	4096	20,727.9	23,605.7	91,668.577	1.14	4.42	4095
Wilkinson	2048	630.481	614.94	1,031.36	0.98	1.64	2047
	4096	9,359.25	10,733.3	26,496.979	1.15	2.83	4095

Table: Running time (in sec.) on a 48-core AMD Opteron 6168 node for four examples.

[3] C. Chen, M. Moreno Maza, and Y. Xie. Cache complexity and multicore implementation for univariate real root isolation. J. of Physics: Conf. Series, 341, 2011.

Parallel multivariate real root isolation

Example	BPAS (parallel)	RealRootIsolate (serial)	Isolate (serial)	Speedup
4-Body-Homog	0.402	0.608	0.382	
Arnborg-Lazard	0.146	0.299	0.066	
Caprasse	0.018	0.14	0.154	7.778
Circles	0.051	0.894	0.814	15.961
Cyclic-5	0.021	0.147	0.206	9.810
Czapor-Geddes-Wang	0.2	0.135	0.184	
D2v10	0.029	0.075	177.999	2.586
D4v5	0.037	0.044	49.09	1.189
Fabfaux	0.192	0.231	0.071	
Katsura-4	0.171	0.416	0.044	
L-3	0.02	0.252	0.12	6.0
Neural-Network	0.029	0.332	0.131	4.517
R-6	0.014	0.048	20.612	3.429
Rose	0.026	0.336	0.599	12.923
Takeuchi-Lu	0.027	0.16	0.031	1.148
Wilkinsonxy	0.023	0.165	0.046	2.0
Nld-10-3	1.249	8.993	707.334	7.20

Table: Running time (in sec.) on a 12-core Intel Xeon 5650 node for BPAS vs. Maple 17 RealRootIsolate vs. C (with Maple 17 interface) Isolate.

Concluding remarks

- The BPAS library is the first polynomial algebra library which emphasizes performance aspects (cache complexity, parallelism) on multi-core architectures
- Its core operations (dense integer polynomial multiplication, real root isolation) outperform their counterparts in recognized computer algebra software (FLINT, Maple)
- Its companion library CUDA Modular Polynomial (CUMODP) has similar goals on GPGPUs www.cumodp.org
- Together, they are designed to support the implementation of polynomial system solvers on hardware accelerators.
- The BPAS library is available in source at www.bpaslib.org

Basic Polynomial Algebra Subprograms