Statement of Authorship

I, Miranda Wood, hereby declare that I am the sole author of this bachelor thesis including all appendices (unless otherwise specified) and that I have not used any sources other than those listed in the bibliography and identified as references.

The Variable Precision Integer Arithmetic toolbox and functions `hex2vpi`, `vpi2hex` are by John D'Errico and Paulo Fonte respectively with sources referenced in the main thesis text. All VPI toolbox functions by D'Errico are included in their own folder with the original license.

Matlab Implementation Guide

An in depth overview of this Matlab implementation is found in the main thesis. What follows is a set-up and practical use guide of the elliptic curve cryptography functions.

Core Files

To use the 'core' (non-VPI) functions one only needs integers a, b, and prime p that describe an elliptic curve in Weierstrass normal form. It is often useful to use `subgsize` to test whether a chosen curve has suitable order subgroups for cryptography (as explained in the thesis). This function outputs a matrix of all points on the curve, so it also useful to find points for testing the background functions `PAdd` and `PMulD`.

To find a suitable G, n, and h simply run `Gnh` with the curve parameters, **saving the outputs to the workspace**. Cryptographic key pairs can now be generated with `btckeyprodall`; d_A the private key corresponding to the public key H_A and similarly d_B to H_B.

The above two functions are combined in `keyprodall` to find keys directly from curve parameters.

Signatures can be created with `sigc` using the parameters found above. Any private key can be used in the place of d_A, but, of course, only the corresponding public key can verify the signature. We input any integer for z, as long as it is not too large for Matlab.

Signatures are verified with `sigv`. It is important that the inputs are the same public parameters that were used to create keys and include the public key corresponding to the relevant private key.

Core Example

We first choose integer curve parameters with a prime p, for example $a = 14$, $b = 9$, $p = 101$.

1. Find the remaining public parameters by running:
   `[G, n, h, a, b, p]=Gnh(14, 9, 101)`
   This run gives `G = 100 14, n = 11, h = 9`.

2. Produce key pairs by running:
   `[da,Ha,db,Hb,G,n,h,a,b,p] = btckeyprodall(G,n,h,a,b,p)`
   This run gives `da = 6, Ha = 17 3, db = 8, Hb = 69 27`.

3. Create a signature for $z = 99$ by the owner of `da`:
   `[r,s,z]=sigc(da,G,n,a,b,p,99)`
   This run gives `r = 9, s = 8, z = 99`.

4. Finally verify the signature with `Ha`:
   `sigv(r,s,z,Ha,G,n,a,b,p)`
   This gives `Signature is valid`.

Steps 1 and 2 can be completed together with:
`[da,Ha,db,Hb,G,n,h,a,b,p] = keyprodall(a, b, p)`.

VPI Files

To use `vpi` objects and relevant operations, one must **add the folder 'VariablePrecisionIntegers', and all subfolders, to the current path**. To create cryptographic keys for realistically large curves, the minimum inputs required are a, b, p, and the curve order N. For this reason, we use well known curves that have published (p, a, b, G, n, h), so practically `GnhVPI` is not needed. One can still use it to find a suitable subgroup G for academic interest by setting `N = n*h`.

Some .MAT files with real curve parameters are included in the folder 'Curve Parameters' for use with the VPI functions. These are the curves `secp192k1`, `secp192r1`, `secp256k1`, and `secp256r1`. After loading these files, our functions work as before.

To input z, or use any custom inputs, one must define them as a `vpi` object. For example, `z=vpi(2 ^ 32 - 1)}`. Even small integers must be input this way; to set $a = 1$ run `a=vpi(1)`. To use a realistic value of z, one can create a SHA-256 hash at `https://www.conversion-tool.com/sha256?lang=en` and input it with `z=hex2vpi('hash')`. Some ready to use examples are available in the file `256bithashes.mat`.

VPI Example

We use the curve `secp192k1` and hashed message `z1`. For completeness, we generate our own subgroup G.

1. Find the remaining public parameters by running:
   `[G, n, h, a, b, p]=GnhVPI(a, b, p, n*h)`

This run gives:
`G = vpi element: (1,1)`
`5460787435686438827241662878248155987243899474896406269859`
`vpi element: (1,2)`
`24594085443699094215169407504846981342206320667133513847 8,`
`n = 627710173538668076383578942306126427195712391520084551 2077,`
`h = 1`.
The original `n` and `h` are replaced.

2. Produce key pairs by running:
   `[da,Ha,db,Hb,G,n,h,a,b,p] = btckeyprodallVPI(G,n,h,a,b,p)`
   For person a, this run gives:
   `da = 575663701486865664701038513841400025235803087789845248320 8,`
   `Ha = vpi element: (1,1)`
   `3987608334893846752628243598090943847559661286361127704594`
   `vpi element: (1,2)`
   `4079573359499319118894430810625331109145221164280576536248`.

3. Create a signature for `z1` by the owner of `da`:
   `[r,s,z]=sigcVPI(da,G,n,z1,a,b,p)`
   This run gives:
   `r = 123331329906691399345686111579107690045400319367976763388 0,`
   `s = 456099071851314156642417776750987345774819029231634023798 2,`
   `z = 609377234261072690643900617828418596017131623005868319270 2`
   `6631652298422836424`.

4. Finally verify the signature with `Ha`:
   `sigvVPI(r,s,z,Ha,G,n,a,b,p)`
   This gives `Signature is valid`.

Either one can start with Step 2, since we have all the parameters, or Steps 1 and 2 can be completed together with:
`[da,Ha,db,Hb,G,n,h,a,b,p] = keyprodallVPI(a, b, p, n*h)`.