Model apply (computeImage) through Abbe and Hopkins models

Created by Li-Hao Yeh, last modified on Feb 06, 2023

Introduction

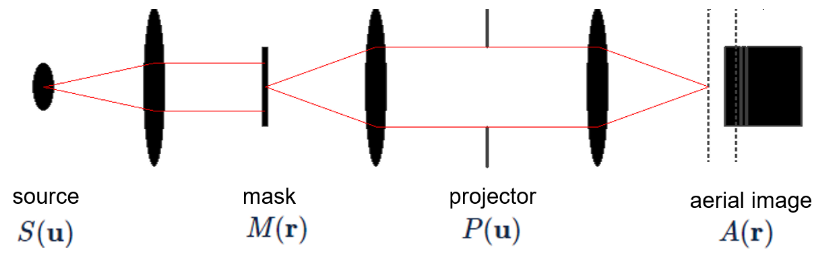
This page tries to provide a more detailed view of the forward and backward flow of the imaging models used in SMO.

SMO has two versions of imaging model, the Abbe and Hopkins model. Both of them describe the imaging formation process of a partially coherent imaging system (for example a microscope or a lithography machine).

Abbe model is more physically intuitive. It describes that the aerial image is the sum of the coherent imaging process (the scattering of each plane wave with the 2D mask) from each source point:

$$A(\mathbf{r}) = \sum_{\mathbf{u}} S(\mathbf{u}) \left| \int \left[\int M(\mathbf{r}') e^{i2\pi \mathbf{u}\cdot\mathbf{r}'} e^{-i2\pi \mathbf{u}'\cdot\mathbf{r}'} d\mathbf{r}' \right] P(\mathbf{u}') e^{i2\pi \mathbf{u}'\cdot\mathbf{r}} d\mathbf{u}' \right|^2, \tag{1}$$

where $\mathbf{r} = (x, y)$ is the 2D spatial coordinate, $\mathbf{u} = (u_x, u_y)$ is the 2D spatial frequency, $A(\mathbf{r})$ is the 2D aerial image, $S(\mathbf{u})$ is the 2D source map, $M(\mathbf{r})$ is the 2D mask image, $P(\mathbf{u})$ is the pupil function (scalar optics). These notations can be found in the following figure:



After some math manipulation, we can obtain Hopkins model from the Abbe model:

$$\begin{aligned} A(\mathbf{r}) &= \textstyle\sum_{\mathbf{u}} S(\mathbf{u}) \left| \int M(\mathbf{u}' - \mathbf{u}) P(\mathbf{u}') e^{i2\pi \mathbf{u}'\cdot\mathbf{r}} d\mathbf{u}' \right|^2 \\ &= \textstyle\sum_{\mathbf{u}} S(\mathbf{u}) \left| \int M(\mathbf{u}'') P(\mathbf{u}'' + \mathbf{u}) e^{i2\pi (\mathbf{u}'' + \mathbf{u})\cdot\mathbf{r}} d\mathbf{u}'' \right|^2 \\ &= \textstyle\sum_{\mathbf{u}} S(\mathbf{u}) \iint M(\mathbf{u}'') M^*(\mathbf{v}'') P(\mathbf{u}'' + \mathbf{u}) P^*(\mathbf{v}'' + \mathbf{u}) e^{i2\pi (\mathbf{u}'' - \mathbf{v}'')\cdot\mathbf{r}} d\mathbf{u}'' d\mathbf{v}'' \\ &= \iint M(\mathbf{u}'') M^*(\mathbf{v}'') \left[\textstyle\sum_{\mathbf{u}} S(\mathbf{u}) P(\mathbf{u}'' + \mathbf{u}) P^*(\mathbf{v}'' + \mathbf{u}) \right] e^{i2\pi (\mathbf{u}'' - \mathbf{v}'')\cdot\mathbf{r}} d\mathbf{u}'' d\mathbf{v}'' \\ &= \iint M(\mathbf{u}'') M^*(\mathbf{v}'') \mathrm{TCC}(\mathbf{u}'', \mathbf{v}'') e^{i2\pi (\mathbf{u}'' - \mathbf{v}'')\cdot\mathbf{r}} d\mathbf{u}'' d\mathbf{v}''. \end{aligned} \tag{2}$$

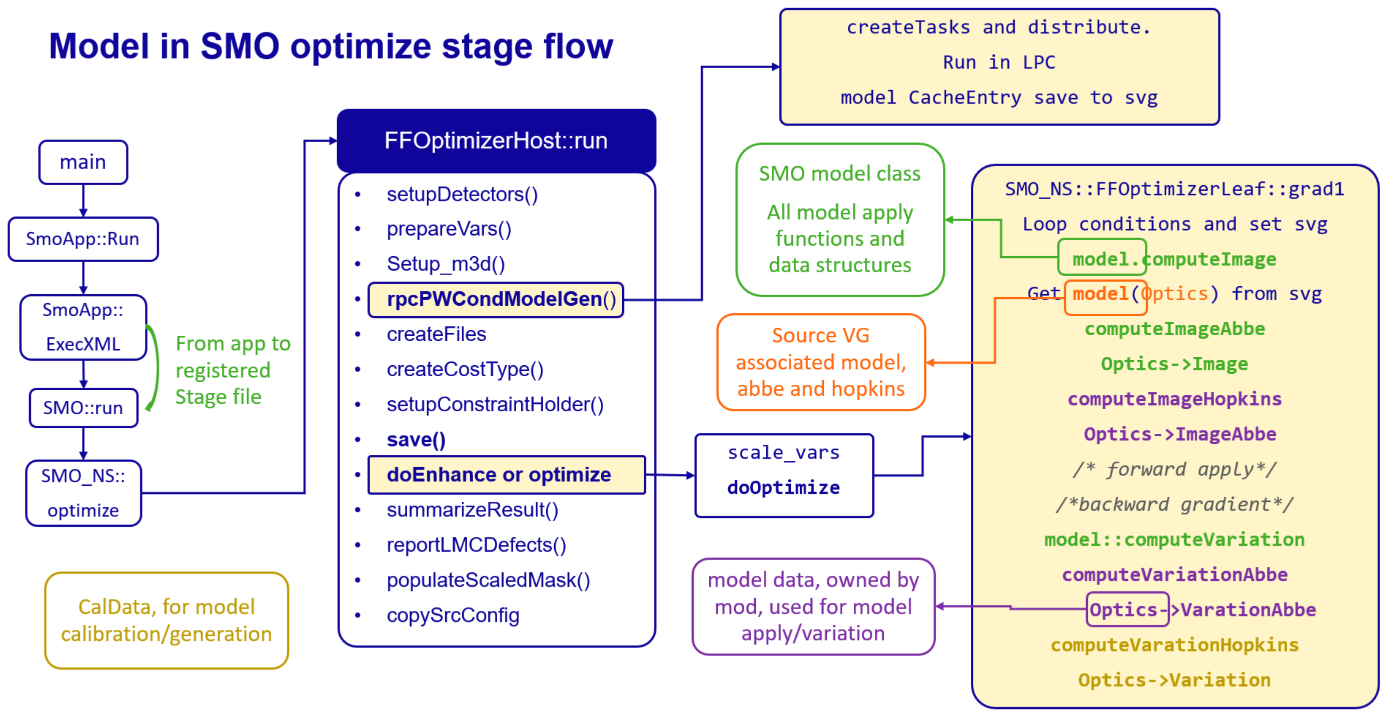
The source map and pupil function are fused into a 4D function called Transmission Cross Coefficient (TCC). Up to now, the Hopkins

$$\begin{aligned} A(\mathbf{r}) &= \iint M(\mathbf{u}'')M^*(\mathbf{v}'')\mathrm{TCC}(\mathbf{u}'',\mathbf{v}'')e^{i2\pi(\mathbf{u}''-\mathbf{v}'')\cdot\mathbf{r}}d\mathbf{u}''d\mathbf{v}'' \\ &\approx \iint M(\mathbf{u}'')M^*(\mathbf{v}'')\left[\sum_i \lambda_i V_i(\mathbf{u}'')V_i^*(\mathbf{v}'')\right]e^{i2\pi(\mathbf{u}''-\mathbf{v}'')\cdot\mathbf{r}}d\mathbf{u}''d\mathbf{v}'' \\ &= \sum_i \lambda_i \left| \int M(\mathbf{u}'')V_i(\mathbf{u}'')e^{i2\pi\mathbf{u}''\cdot\mathbf{r}}d\mathbf{u}''\right|^2. \end{aligned} \tag{3}$$

With the basic understanding of these two models, we proceed to talk about SMO and MOD's implementation on these two models.

When to call Model::computeImage

computeImage is in charge of computing aerial image with either Abbe or Hopkins model. The following graph is first created by BC RD (original slides) to summarize how SMO arrives at model::computeImage. It starts with SMO::run to run an optimize or opc stage. In the host of the stage (FFOptimizerHost::run in ffopthost.cc), the TCC is generated in rpcPWCondModelGen(). For Abbe model in SMO's implementation, a single-source-point TCC is needed to account for polarization effect (link). For Hopkins model, this TCC captures the full optics effect (source, pupil, polarization, etc.). After TCC is built, the program proceeds to the leaf node to compute the aerial image in the grad1 function (FFOptimizerLeaf::grad1 in ffoptleaf.cc). Depending on the model in the sourceVG, the program will compute aerial image with Abbe (for example, freesourcevar with srcfix=0, any opc or optimize stage) or Hopkins (for example, dumbsourcevar after modelgen1200) model. The aerial image is later used to compute the resist image (cost_optics / cost_resist), costs, and gradients.



Model::computeImageAbbe

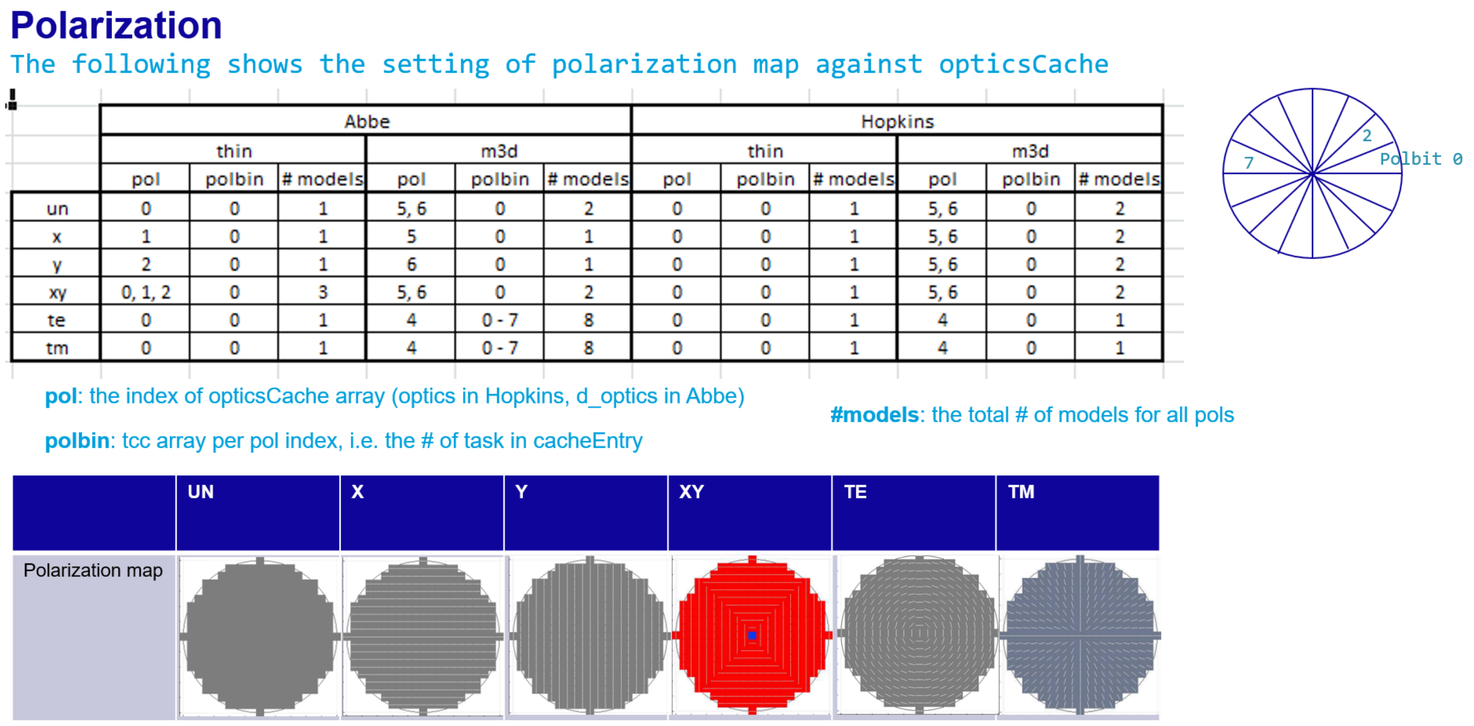
SMO part of work

This is SMO's implementation of Abbe model. This function computes the aerial image, $A(\mathbf{r})$ (`ImageTemplate<float> ai`), from given source map, $S(\mathbf{u})$ (`float* sw = &sm->d_srctmp()[0]`), mask image $M(\mathbf{r})$ (`complex<float>* mi = mvg→thinbuffers(role).mi()`, mask 2D), and pupil function $P(\mathbf{u})$ (`complex<float>* pf = lvg->getfilter()`). The main structure of the function can be summarized as following:

- Retrieve relevant parameters from sourcevg (`svg`), maskvg (`mvg`), and lensvg (`lvg`)
- Create buffers for ai computation
- Loop through different polarization options and sum them together. Within each loop, it does:
    - Look up the corresponding TCC (model) for a specific polarization option (`OpticsObj opticsobj = sm-`

```
        sw, pf, patch.w, patch.h, patch.x, patch.y, &wmi))
```

To understand how the index in the loop corresponds to the polarization option, BC RD again has a nice table (original slides) to summarize it:



Polarization

The following shows the setting of polarization map against opticsCache

	Abbe						Hopkins					
	thin			m3d			thin			m3d		
	pol	polbin	# models	pol	polbin	# models	pol	polbin	# models	pol	polbin	# models
un	0	0	1	5, 6	0	2	0	0	1	5, 6	0	2
x	1	0	1	5	0	1	0	0	1	5, 6	0	2
y	2	0	1	6	0	1	0	0	1	5, 6	0	2
xy	0, 1, 2	0	3	5, 6	0	2	0	0	1	5, 6	0	2
te	0	0	1	4	0 - 7	8	0	0	1	4	0	1
tm	0	0	1	4	0 - 7	8	0	0	1	4	0	1

pol: the index of opticsCache array (optics in Hopkins, d_optics in Abbe)

polbin: tcc array per pol index, i.e. the # of task in cacheEntry

#models: the total # of models for all pols

	UN	X	Y	XY	TE	TM
Polarization map						

For example, if we have xy polarization selected in SMO, SMO will then build 3 TCCs (unpolarized, x-polarized, and y-polarized) and then loop these 3 TCCs in computeImageAbbe to compute the aerial image.

MOD part of work

So far, we have understood the structure of computeImageAbbe. The core computation of the aerial image lies within optics→ImageAbbe (app/litho/model_data/optics.cc), so let's dive into this function. Tracing this piece of code, we find that ImageAbbe calls ImageAbbeTask::compute (inherited from CompTask, multithreading process) to compute the aerial image per source point in parallel. The following is the core part of the computation for a thin mask:

ImageAbbeTask::compute

```

for (int is = data->d_iv1; is < data->d_iv2; is++) {
    if (!(data->d_m1 & 1 << (is - data->d_iv1)))
        continue;
    const SourcePoint& p = (*d_srcpts)[is];
    if (!d_m3d) {
        optics_shift(d_mx + off, t.buf3, p.xsf, p.ysf, d_size, d_rowskip);
        fftwf_execute_dft(t.fplan2, (fftwf_complex*)t.buf3, (fftwf_complex*)t.buf3);
        if (d_pf)
            optics_evec(d_pf, t.buf3, t.buf3, s2);
        for (int jv = 0; jv < d_nterm; jv++) {
            int iv = jv * d_nsrc + p.isrc;
            float eval = sw[is] * tcc->eval_flt[iv] * d_scale;
            const complex<float>* evec = &tcc->evec_flt[iv * s2];
            optics_evec(evec, t.buf3, t.buf2, s2);
            fftwf_execute(t.iplan);
            optics_eval(t.buf2, eval, (float*)t.buf1, d_size, d_guard);
```

To explain each line of code:
7868: loop through the source points distributed to this thread
7869: if the source point is labeled as skip (e.g. weight = 0), then skip it
7871: retrieve the information for a specific source point
7872: enter this block if it is a thin-mask model
7873: optics_shift multiply the plane wave ($e^{i2\pi\mathbf{u}\cdot\mathbf{r}}$, `p.xsf`, `p.ysf` are the exponent factors in x and y directions) on the 2D mask (`d_mx`) and save the result in `t.buf3` (buffer for FFT struct)
7874: conduct a 2D DFT on the array saved in `t.buf3` and save the result in `t.buf3`.
7876: point-wise mutiply the pupil function (`d_pf`) with `t.buf3` and save the result in `t.buf3`
7877: loop through different TCC terms
7879: compute the overall weight for this aerial image, which is source weight * eigenvalue * some scaling factor
7880: retrieve the eigenvector of the TCC
7881: point-wise mutiply the eigenvector (`evec`) with `t.buf3` and save the result in `t.buf2`
7882: execute a 2D inverse DFT on `t.buf2` and save the result in `t.buf2`
7883: scale the image in `t.buf2` with the overall weight (`eval`) and save the result in `t.buf1`
End: Accumulate the result in `t.buf1` to the array `img` and return the result.

From the above operations, we can write out the actual mathematical form of the Abbe model in Tachyon as:

$$A(\mathbf{r}) = \sum_{\text{pol},\mathbf{u},i} S_{\text{pol}}(\mathbf{u})\lambda_{\text{pol},i}\left|\mathcal{F}^{-1}\left\{\mathcal{F}\left\{M(\mathbf{r}')e^{i2\pi\mathbf{u}\cdot\mathbf{r}'}\right\}(\mathbf{u}'')P(\mathbf{u}'')V_{\text{pol},i}(\mathbf{u}'')\right\}(\mathbf{r})\right|^2, \tag{4}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the forward and inverse DFT. This equation looks almost the same as (1) except for the presence of eigenvectors of the TCC. From the look of this equation, each source point's forward simulation is done by applying the TCC (built with only DC component in the source map) on the frequency-shifted mask.

Further explanation on Tachyon's Abbe model

To explain the difference between (1) and (4), we need to answer the following questions:

1. Why do we have TCC in the Abbe model? Isn't it specifically for Hopkins model?
2. If we do need a TCC, why do we only need the TCC from the central source point?

The answer to the first question is that SMO leverages the handling of polarization effect from TCC in Tachyon. In Tachyon, a vector optics version of the TCC is used, which is defined as

$$\text{TCC}(\mathbf{u}'',\mathbf{v}'') = \sum_{\mathbf{u}}\sum_{i=x,y}\sum_{j=x,y} E_i(\mathbf{u})E_j^*(\mathbf{u})P_i(\mathbf{u}+\mathbf{u}'')P_j^*(\mathbf{u}+\mathbf{v}''), \tag{5}$$

where $E_i(\mathbf{u})$ is the electric fields in x and y direction from the source, $P_i(\mathbf{u})$ is the corresponding pupil functions for x and y polarized light. Since SMO optimizes source and the pupil function, SMO owns its own source and pupil functions. When building the TCC in MOD's code, SMO will provide a source map with $E_i(\mathbf{u}) = 0$ when $\mathbf{u} \neq 0$ and asks the pupil function to be aberration-free. In (4), the scalar optics pupil function and source map are owned by SMO. They are used on top of the TCC built with aberration-free vectorial pupil and the central source point. For further explanation, see here.

The second question is how we can construct the aerial image only from TCC of the central source point. Consider the TCC of the central source point:

$$\text{TCC}_0(\mathbf{u}'',\mathbf{v}'') = \sum_{i=x,y}\sum_{j=x,y} E_iE_j^*P_i(\mathbf{u}'')P_j^*(\mathbf{v}'').$$

If we have a TCC for an off-axis source point $\text{TCC}_1(\mathbf{u}'',\mathbf{v}'') = \sum_{i=x,y}\sum_{j=x,y} E_iE_j^*P_i(\mathbf{u}_0+\mathbf{u}'')P_j^*(\mathbf{u}_0+\mathbf{v}'')$, and want to construct its aerial image, from (2), we can write

$$= \iint M(\mathbf{u}'')M^*(\mathbf{v}'')\mathrm{TCC}_0(\mathbf{u}''+\mathbf{u}_0,\mathbf{v}''+\mathbf{u}_0)e^{i2\pi(\mathbf{u}''-\mathbf{v}'')\cdot\mathbf{r}}d\mathbf{u}''d\mathbf{v}''$$
$$= \iint M(\mathbf{u}''-\mathbf{u}_0)M^*(\mathbf{v}''-\mathbf{u}_0)\mathrm{TCC}_0(\mathbf{u}'',\mathbf{v}'')e^{i2\pi(\mathbf{u}''-\mathbf{v}'')\cdot\mathbf{r}}d\mathbf{u}''d\mathbf{v}''.$$

From this equation, we can compute the aerial image of an off-axis source point using the TCC from the central source point. If we did so for all the source points and decompose the TCC of the central source point into eigenvectors, then we arrive equation (4).

Model::computeVariationAbbe

SMO part of work

This function computes the gradient of the mask (`mvg->thinbuffers(role).gi()`), source map (`float* sp`), and pupil function (`complex<float>* gpf`) from the gradient of the aerial image (`ImageTemplate<float> wa`) and the gradient of the mask (`ImageTemplate<complex<float>> wm`) generated from a resist/optics model. The main structure of the function can be summarized as following:

- Retrieve relevant parameters from sourcevg (`svg`), maskvg (`mvg`), and lensvg (`lvg`).
- This includes the variables used in the forward model (mask, source, pupil function) and the buffer for the gradient of the targeted variables.
- Loop through different polarization options for the gradient of all variables and sum them together. Within each loop, it does:
    - Look up the corresponding TCC (model) for a specific polarization option (`OpticsObj opticsobj = sm->getmodel(d_data, patchsize, pol, mxm3d)`)
    - Adjust source weight and reset source gradient according to the polarization option (`sm->getweight(pol, -1, sw, sp)`)
    - Call MOD's VariationAbbe function to compute the gradient per polarization option (`optics->VariationAbbe(gibuf, mi, temp, wa.pix, wf, sw, sp, gpf, pfp, patch.w, patch.h, patch.x, patch.y)`)
    - Accumulate the variation of the source and pupil function to their corresponding variables (`sm->setvariation(pol, -1, sp)`, `lvg->setvariation(pfp)`)

MOD part of work

Similar to the forward model, the core computation of the variation happens in MOD's code base. The math derivation of the variation is explained in this documentation written by Rafe (Rafe_variation.pdf). Once we understand the forward model and the math of the variation, the code of the variation step becomes easier to parse. It uses the same CompTask framework to conduct multi-threading process. The optics→VariationAbbe calls VariationAbbeTask::compute to conduct the computation. The following is the core part of the computation of the variation:

VariationAbbeTask::compute

```

for (int is = data->d_iv1; is < data->d_iv2; is++) {
    int m1 = data->d_m1 & 1 << (is - data->d_iv1);
    int m2 = data->d_m2 & 1 << (is - data->d_iv1);
    if (!m1 && !m2)
        continue;
    const SourcePoint& p = (*d_srcpts)[is];
    if (!d_m3d) {
        optics_shift(d_mx + off, t.buf3, p.xsf, p.ysf, d_size, d_rowskip);
        fftwf_execute_dft(t.fplan2, (fftwf_complex*)t.buf3, (fftwf_complex*)t.buf3);
    } else {
        optics_shift(d_mx + off, mx, p.xsf, p.ysf, d_size, d_rowskip);
        optics_shift(d_my + off, my, p.xsf, p.ysf, d_size, d_rowskip);
        fftwf_execute_dft(t.fplan2, (fftwf_complex*)mx, (fftwf_complex*)mx);
        fftwf_execute_dft(t.fplan2, (fftwf_complex*)my, (fftwf_complex*)my);
    }
```

```
8758                const complex<float>* evec = &tcc->evec_flt[iv * s2];
8759                optics_evec(evec, t.buf3, t.buf2, s2);
8760            } else {
8761                const complex<float>* evecX = &tcc->evec_flt[iv * s2 * 2];
8762                const complex<float>* evecY = evecX + s2;
8763                optics_evec(evecX, evecY, mx, my, t.buf2, s2);
8764            }
8765            if (d_pf)
8766                optics_evec(d_pf, t.buf2, t.buf2, s2);
8767            fftwf_execute(t.iplan);
8768            if (m2) {
8769                float eval = tcc->eval_flt[iv] * d_scale;
8770                dsp[is] += optics_eval_wt(t.buf2, eval, wt + off, d_size, d_guard, d_rowskip);
8771            }
8772            if (m1) {
8773                optics_conj_wt(t.buf2, wt + off, d_size, d_guard, d_rowskip);
8774                fftwf_execute(t.fplan2);
8775                float eval = sw[is] * tcc->eval_flt[iv] * d_scale;
8776                if (!d_m3d) {
8777                    const complex<float>* evec = &tcc->evec_flt[iv * s2];
8778                    // pupil variation
8779                    if (d_gpf) {
8780                        optics_evec(evec, t.buf3, buf5, s2);
8781                        optics_evec_inv(t.buf2, buf5, buf5, d_size);
8782                        optics_mul_accum(buf5, eval, gpf, s2);
8783                    }
8784                    // mask variation
8785                    optics_evec_inv(evec, t.buf2, t.buf2, d_size);
8786                    if (d_pf)
8787                        optics_evec_inv(d_pf, t.buf2, t.buf2, d_size);
8788                    fftwf_execute(t.iplan);
8789                    optics_eval_shift(t.buf2, eval, img, p.xsf, p.ysf, d_size);
```

To explain the key lines of this code block:
8741: m1 is the flag on whether to compute the gradient of the mask and pupil function
8742: m2 is the flag on whether to compute the gradient of the source
Forward evaluation
8747-8748: optics_shift multiply the plane wave on the 2D mask (`d_mx`), a DFT is conducted on the result, and the result is saved in `t.buf3`
8759: point-wise mutiply the eigenvector (`evec`) with `t.buf3` and save the result in `t.buf2`
8766: point-wise mutiply the pupil function (`d_pf`) with `t.buf2` and save the result in `t.buf2`
8767: execute a 2D inverse DFT on `t.buf2` and save the result in `t.buf2`
Source gradient
8770: accumulate $\sum_{\mathbf{r}} \text{eval} \times d\text{cost}/dA(\mathbf{r}) \times |\text{t.buf2}(\mathbf{r})|^2$ into the gradient of the specific source point
Pupil and mask gradient
8773: compute $d\text{cost}/dA(\mathbf{r}) \times \text{conj}\{\text{t.buf2}(\mathbf{r})\}$ and save the result to `t.buf2`
8774: conduct a 2D DFT on `t.buf2` and save the result in `t.buf2`
Pupil gradient
8780: point-wise mutiply the eigenvector (`evec`) with `t.buf3` and save the result in `buf5`
8781: flip `t.buf2` (reversed the coordinate), point-wise multiply with `buf5`, and save the result in `buf5`
8782: scale `buf5` with the eigenvalue and save it to `gpf`
Mask gradient

8789. multiply the eigenvalue and the plane wave factor to `t.buf2` and accumulate the result into `img` pointer (before updating the mask, the real part of this gradient will be taken.)

With all the math operations understood in the code, let's summarize these operations in the mathematical language. At the end of line 8767, `t.buf2` represents an electric field per source point, eigen-state, and polarization state and can be written out as

$$E_{\mathrm{pol},\mathbf{u},i}(\mathbf{r}) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{M(\mathbf{r}')e^{i2\pi\mathbf{u}\cdot\mathbf{r}'}\right\}(\mathbf{u}'')P(\mathbf{u}'')V_{\mathrm{pol},i}(\mathbf{u}'')\right\}(\mathbf{r}).$$

With line 8770 and further accumulation process, we can summarize the gradient to a single source point to be

$$\frac{d\mathrm{cost}}{dS(\mathbf{u})} = \sum_{\mathrm{pol},i}\lambda_{\mathrm{pol},i}\int d\mathbf{r}\frac{d\mathrm{cost}}{dA(\mathbf{r})}|E_{\mathrm{pol},\mathbf{u},i}(\mathbf{r})|^2$$

With line 8773 and 8774, we update what is saved in `t.buf2` as a new function summarized as

$$B_{\mathrm{pol},\mathbf{u},i}(\mathbf{u}'') = \mathcal{F}\left\{\frac{d\mathrm{cost}}{dA(\mathbf{r})}E^*_{\mathrm{pol},\mathbf{u},i}(\mathbf{r})\right\}(\mathbf{u}'').$$

With line 8780-8782, we can summarize the gradient to the pupil function as (corresponding to equation 15 in Rafe's derivation)

$$\frac{d\mathrm{cost}}{dP(\mathbf{u}'')} = \sum_{\mathrm{pol},\mathbf{u},i}S(\mathbf{u})\lambda_{\mathrm{pol},i}B_{\mathrm{pol},\mathbf{u},i}(-\mathbf{u}'')M(\mathbf{u}''-\mathbf{u})V_{\mathrm{pol},i}(\mathbf{u}''),$$

where $M(\mathbf{u}''-\mathbf{u})$ is the result saved in `t.buf3`.

With line 8785-8789, we can write the gradient to the mask function as

$$\frac{d\mathrm{cost}}{dM(\mathbf{r})} = \sum_{\mathrm{pol},\mathbf{u},i}S(\mathbf{u})\lambda_{\mathrm{pol},i}\mathcal{F}^{-1}\left\{B_{\mathrm{pol},\mathbf{u},i}(\mathbf{u}'')P(-\mathbf{u}'')V_{\mathrm{pol},i}(-\mathbf{u}'')\right\}(\mathbf{r})e^{i2\pi\mathbf{u}\cdot\mathbf{r}}. \tag{6}$$

This gradient is only the contribution from the $d\mathrm{cost}/d(M(\mathbf{r})e^{i2\pi\mathbf{u}\cdot\mathbf{r}})$. A full gradient should include the contribution from its counterpart $d\mathrm{cost}/d(M(\mathbf{r})e^{i2\pi\mathbf{u}\cdot\mathbf{r}})^*$. According to Rafe's deriviation, we could see this other part of the contribution is exactly the complex conjugate of the computed part. Summing these two parts together, we have the final mask gradient to be the real part of the (6).

Model::computeImageHopkins

When calling this function, we input a mask image $M(\mathbf{r})$ and use the pre-built TCC to compute the aerial image, $A(\mathbf{r})$. The effects of source, aberration, and polarization are all lumped into the TCC already. The implementation of the Hopkins model in SMO part has very similar structure to the case of the Abbe model. Except in this case, the image model apply is done once without looping over different polarization options and there is no need to adjust the source weight. Hence, we will focus the study on MOD part of the code. The core computation of the Hopkins model happens in ImageTask::compute in optics.cc (from the call of optics→Image<complex<float>,float>). The following is the code block for this computation:

ImageTask::compute

```

    for (int iv = data->d_iv1; iv < data->d_iv2; iv++) {
        if (iv >= 0) {
            const complex<float>* evec = &tcc->evec_flt[iv * d_size * d_size];
            optics_evec(evec, d_fft->buf1, t.buf2, d_size * d_size);
```

```
2498            //
2499            //
2500            //
2501            fftwf_execute(t.iplan);
2502            //
2503            //
2504            //
2505            //
2506            float eval = tcc->eval_flt[iv] * d_scale;
2507            if (!ai_formation_us) {
2508                optics_eval(t.buf2, eval, img, d_size, d_guard);
```

If you are familiar with the functions for math operations done in the Abbe model, this code block can be understood more easily. There are only three math operations involved.
2494: The first is to multiply `d_fft->buf1` with the eigenvector, `evec`, and save the result to `t.buf2`. (Note that before coming to this function call, d_fft->buf1 is configured to store the Fourier transform of the mask image in Optics::ImageFloat)
2501: The second is to conduct an inverse Fourier transform on `t.buf2`.
2508: The third operation is to accumulate the $\lambda_i\,|\mathrm{t.\,buf2}(\mathbf{r})|^2$ into the aerial image intensity, `img`.

With this understanding, we can summarize the math operation of the Hopkins model in Tachyon as:

$$A(\mathbf{r}) = \sum_i \lambda_i \left|\mathcal{F}^{-1}\left\{\mathcal{F}\{M(\mathbf{r}')\}(\mathbf{u})V_i(\mathbf{u})\right\}(\mathbf{r})\right|^2, \tag{7}$$

which is exactly the same as (3).

Model::computeVariationHopkins

This function computes the gradient of the mask from the gradient of the aerial image and the gradient of the mask generated from a resist/optics model. computeVariationHopkins shares the same structure as in computeVariationAbbe. The SMO part of the work is mainly to retrieve and prepare the variables for the gradient computation. We will again focus on the MOD part of the work. The core computation of the variation of Hopkins model happens in VariationTask::compute in optics.cc (from the call of optics→Variation). The following is the code block for this computation:

VariationTask::compute

```

for (int iv = data->d_iv1; iv < data->d_iv2; iv++) {
        if (iv < 0)
            continue;
        const complex<float>* evec = &tcc->evec_flt[iv * d_size * d_size];
        for (int k = 0; k < d_size * d_size; k++)
            t.buf2[k] = evec[k] * d_fft->buf1[k];
        fftwf_execute(t.iplan);
        // use wt only within [ g, g, s - g, s - g ] bbox
        for (int j = 0, k = 0; j < d_size; j++) {
            if (j < d_guard || j >= d_size - d_guard) {
                memset(&t.buf2[k], 0, d_size * sizeof(t.buf2[0]));
                k += d_size;
                continue;
            }
            memset(&t.buf2[k], 0, d_guard * sizeof(t.buf2[0]));
            k += d_guard;
6063                }
6064                memset(&t.buf2[k], 0, d_guard * sizeof(t.buf2[0]));
6065                k += d_guard;
6066            }
6067            fftwf_execute(t.fplan2);
6068            for (int j = 0, k = 0; j < d_size; j++)
6069                for (int i = 0; i < d_size; i++, k++) {
6070                    int kk = (j ? d_size - j : j) * d_size + (i ? d_size - i : i);
6071                    t.buf2[k] = evec[kk] * t.buf2[k];
6072                }
6073            fftwf_execute(t.iplan);
6074            float eval = tcc->eval_flt[iv] * d_scale;
6075            for (int k = 0; k < d_size * d_size; k++)
6076                img[k] += eval * t.buf2[k];
6077        }
```

To explain the key lines of this code block:
6049: multiply `d_fft->buf1` with the eigenvector, `evec`, and save the result to `t.buf2`. (Note that before coming to this function call, `d_fft->buf1` is configured to store the Fourier transform of the mask image in Optics::Variation)
6050: conduct an inverse Fourier transform on the `t.buf2`.
6062: multiply the gradient of ai with the complex conjugate of `t.buf2` and save to `t.buf2`
6067: conduct a Fourier transform on the `t.buf2`
6071: multiply `t.buf2` with the coordinate-flipped eigenvector
6076: accumulate the gradient from a single TCC term into img.

Since (7) is a mathematically-simplified version of (4), the gradients share a similar structure. We can summarize the gradient of the mask from a Hopkins model using the implementation above as:

$$\frac{d\text{cost}}{dM(\mathbf{r})} = \sum_i \lambda_i \mathcal{F}^{-1}\left\{B_i(\mathbf{u}'')V_i(-\mathbf{u}'')\right\}(\mathbf{r}),$$

where

$$B_i(\mathbf{u}'') = \mathcal{F}\left\{\frac{d\text{cost}}{dA(\mathbf{r})}E_i^*(\mathbf{r})\right\}(\mathbf{u}''),$$

$$E_i(\mathbf{r}) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{M(\mathbf{r}')\right\}(\mathbf{u}'')V_i(\mathbf{u}'')\right\}(\mathbf{r}).$$

No labels